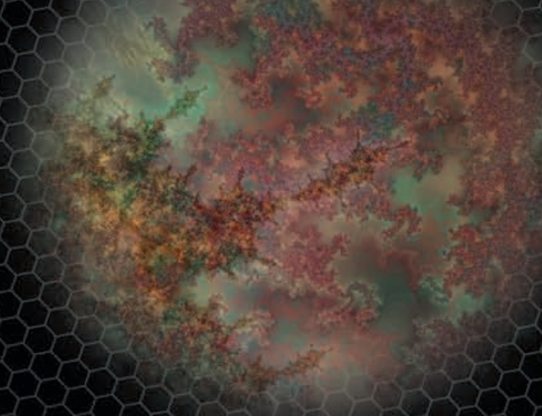
THE EXPERT'S VOICE® IN OPEN SOURCE



Foundations of Python Network Programming

The comprehensive guide to building network applications with Python

SECOND EDITION

Brandon Rhodes and John Goerzen

Apress®

Foundations of Python Network Programming

The comprehensive guide to building network applications with Python

Second Edition

Brandon Rhodes
John Goerzen

Apress®

Foundations of Python Network Programming: The comprehensive guide to building network applications with Python

ISBN-13 (pbk): 978-1-4302-3003-8

ISBN-13 (electronic): 978-1-4302-3004-5

Printed and bound in the United States of America (POD)

President and Publisher: Paul Manning
Lead Editor: Frank Pohlmann
Development Editor: Matt Wade
Technical Reviewer: Michael Bernstein
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Laurin Becker
Copy Editors: Mary Ann Fugate and Patrick Meador
Compositor: MacPS, LLC
Indexer: Potomac Indexing, LLC
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/info/bulksales`.

The source code for this book is available to readers at `www.apress.com`.

To the Python community
for creating a programming language, libraries, and packages
that are freely written and freely gifted
from one programmer to another

To small Persephone-cat for keeping me warm
while revising chapters late at night

And, most of all, to my Jackie

Contents at a Glance

Contents

About the Authors



Brandon Craig Rhodes has been an avid Python programmer since the 1990s, and a professional Python developer for a decade. He released his PyEphem astronomy library in the same year that Python 1.5 was released, and has maintained it ever since.

As a writer and speaker, Brandon enjoys teaching and touting Python, whether as the volunteer organizer of Python Atlanta or on stage at conferences like PyCon. He was editor of the monthly *Python Magazine*, was pleased to serve as technical reviewer for the excellent *Natural Language Processing with Python*, and has helped several open source projects by contributing documentation.

Today Brandon operates the Rhodes Mill Studios consultancy in Atlanta, Georgia, which provides Python programming expertise and web development services to customers both local and out-of-state. He believes that the future of programming is light, concise, agile, test-driven, and enjoyable, and that Python will be a big part of it.



John Goerzen is an accomplished author, system administrator, and Python programmer. He has been a Debian developer since 1996 and is currently president of Software in the Public Interest, Inc. His previously published books include the *Linux Programming Bible*, *Debian Unleashed*, and *Linux Unleashed*.

About the Technical Reviewer

■ **Michael Bernstein** is a web designer and developer, specializing in usable, simple, standards-based web applications, living in Albuquerque, New Mexico.

Acknowledgements

This book owes its very existence to John Goerzen, whose work in writing the first edition of *Foundations of Python Network Programming* indeed provided the foundation on which this volume has been built. The excellent example he set by supplying complete, working example programs has guided me at every step. Where his examples were not obsolete, I have worked to retain his source code so that it can benefit another generation of readers.

The editorial team at Apress provided ample support during this experience—my first attempt at revising something the length of an entire book—and the quality of the result is in large part thanks to Laurin Becker's gentle encouragement, Michael R. Bernstein's very knowledgeable technical reviews, and Matt Wade's holding the rudder to keep each chapter on course. Michael's reviews, in particular, were a model of what an author needs: frequent encouragement when a chapter has gone well, tips and links to more information when coverage of a topic is sketchy, and frank dismay when part of a chapter has gone off the rails. Several parts of this book that will please readers will do so because their first draft was not adequate, and Michael suggested the direction in which the chapter needed to move instead.

And, of course, the copy editors and layout people all did much work as well, and I want to thank Mary Ann Fugate in particular for imposing her good taste about when to use "which" and when to use "that," which (that?) has produced much smoother English.

Every reader of this book should join me in thanking the Python core developers and the community that has grown up around Python for every single tool, routine, and function referenced in this book. And as John Goerzen did in the first edition's acknowledgments, I want to express gratitude to the early generations of programmers like Richard Stallman, who demonstrated that programming could be an open, happy, and cooperative discipline that did not impose the physical world's economics of scarcity onto the world of freely copied programs. To those who prefer more negative forms of protest, I offer Joss Whedon's mantra about creativity: "The greatest expression of rebellion is joy."

And, finally, I would like to thank my mother for letting me spend enough time in front of the computer when I was growing up, and my father for raising me in a house with shelves of books about Unix. He chose an AT&T 3B1 as our home computer. While other students in grade school were learning about the abysmal world of DOS, I was learning about awk, C, and multi-processing—background that prepared me to appreciate Python's beauty the moment I saw it.

Brandon Craig Rhodes
Midtown Atlanta
19 November 2010

Introduction

You have chosen an exciting moment in computing history to embark on a study of network programming. Machine room networks can carry data at speeds comparable to those at which machines access their own memory, and broadband now reaches hundreds of millions of homes worldwide. Many casual computer users spend their entire digital lives speaking exclusively to network services; they are only vaguely aware that their computer is even capable of running local applications.

This is also a moment when, after 20 solid years of growth and improvement, interest in Python really seems to be taking off. This is different from the trajectory of other popular languages, many of which experience their heyday and go into decline long before the threshold of their third decade. The Python community is not only strong and growing, but its members seem to have a much better feel for the language itself than they did a decade ago. The advice we can share with new Python programmers about how to test, write, and structure applications is vastly more mature than what passed for Pythonic design a mere decade ago.

Both networking and Python programming are large topics, and their intersection is a rich and fertile domain. I wish you great success! Whether you just need to connect to a single network port, or are setting out to architect a complex network service, I hope that you will remember that the Internet is an ecosystem that remains healthy so long as individual programmers honor public protocols and support interoperability so that solutions can grow, compete, and thrive.

Writing even the simplest network program inducts you into the grand tradition started by the inventors of the Internet, and I hope you enjoy the tools and the power that they have placed in our hands. I like the encouragement that John Goerzen, the author of the first edition of this book, gave his readers in his own introduction: "I want this to be your lab manual—your guide for inventing things that make the Internet better."

Assumptions

This book assumes that you know how to program in Python, but does not assume that you know anything about networking. If you have used something like a web browser before, and are vaguely aware that your computer talks to other computers in order to display web pages, then you should be ready to start reading this book.

This book targets Python versions 2.5, 2.6, and 2.7, and in the text I have tried to note any differences that you will encounter between these three versions of Python when writing network code.

As of this writing, the Python 2 series is still the workaday version of the language for programmers who use Python in production. In fact, the pinnacle of that line of language development—Python 2.7—was released just a few months ago, and a second bugfix release is now in testing. Interest in the futuristic Python 3 version of the language is still mostly limited to framework authors and library maintainers, as they embark on the community's several-year effort to port our code over to the new version of the language.

If you are entirely new to programming, then an Amazon search will suggest several highly rated books that use Python itself to teach you the basics. A long list of online resources, some of which are complete e-books, is maintained at this link: wiki.python.org/moin/BeginnersGuide/NonProgrammers.

If you do know something about Python and programming but run across unfamiliar syntax or conventions in my program listings, then there are several sources of help. Re-reading the Python Tutorial—the document from which I myself once learned the language—can be a great way to review all of the language's basic features. Numerous books are, of course, available. And asking questions on Stack Overflow, a mailing list, or a forum might help you answer questions that none of your printed materials seem to answer directly.

The best source of knowledge, however, is often the community. I used Python more or less alone for a full decade, thinking that blogs and documentation could keep me abreast of the latest developments. Then a friend convinced me to try visiting a local Python users group, and I have never been the same. My expertise started to grow by leaps and bounds. There is no substitute for a real, live, knowledgeable person listening to your problem and helping you find the way to a solution.

Networking

This book teaches network programming by focusing on the Internet protocols—the kind of network in which most programmers are interested these days, and the protocols that are best supported by the Python Standard Library. Their design and operation is a good introduction to networking in general, so you might find this book useful even if you intend to target other networks from Python; but the code listings will be directly useful only if you plan on speaking an Internet protocol.

The Internet protocols are not secret or closed conventions; you do not have to sign non-disclosure agreements to learn the details of how they operate, nor pay license fees to test your programs against them. Instead, they are open and public, in the best traditions of programming and of computing more broadly. They are defined in documents that are each named, for historical reasons, a Request For Comments (RFC), and many RFCs are referred to throughout this book.

When an RFC is referenced in the text, I will generally give the URL to the official copy of each RFC, at the web site of the Internet Engineering Task Force (IETF). But some readers prefer to look up the same RFCs on `faqs.org` since that site adds highlighting and hyperlinks to the text of each RFC; here is a link to their archive, in case you might find a richer presentation helpful: `www.faqs.org/rfcs/`.

Organization

The first of this book's four parts is the foundation for all of the rest: it explains the basic Internet protocols on which all higher forms of communication are built. If you are writing a network client, then you can probably read Chapters 1 through 6 and then jump ahead to the chapter on the protocol that interests you. Programmers interested in writing servers, however, should continue on through Chapter 7—and maybe even Chapter 8—before jumping into their specific protocol.

The middle parts of the book each cover a single big topic: the second part covers the Web, while the third looks at all of the different protocols surrounding e-mail access and transmission. It is upon reaching its fourth part that this book finally devolves into miscellany; the chapters bounce around between protocols for interacting with command prompts, transferring files, and performing remote procedure calls.

I want to draw particular attention to Chapter 6 and the issue of privacy online. For too many years, encryption was viewed as an exotic and expensive proposition that could be justified only for information of the very highest security. But with today's fast processors, SSL can be turned on for nearly any service without your users necessarily seeing any perceptible effect. And modern Python libraries make it easy to establish a secure connection! Become informed about SSL and security, and consider deploying it with all externally facing services that you write for public use.

Program Listings

Indentation is always a problem when putting Python programs in a book, because when a program listing is broken between pages, it can be difficult to determine whether the indentation level happened to change at the page break. The editors at Apress were very supportive when I offered an idea: we have inserted light gray chevrons to mark each four-space level of indentation.

We used the » symbol because it is not a valid character in a Python program, and therefore—we hope—readers will not be confused and try to insert it in their actual listings! Everywhere that you see the gray double chevron, understand that the actual code listing simply contains a space, and that the chevrons are there to make the number of spaces easier for you to count. Please let us know whether you indeed find this innovation more or less confusing than a traditional, unadorned program listing.

To learn a new programming concept, the best thing you can often do is to experiment. To encourage this, Apress makes the source code for their books' program listings freely available for download at `apress.com`. Please take advantage of this and transform the listings on these printed pages into living programs loaded into your text editor. You are even free to use the code in your own projects!

In the source bundle, I am providing a special treat: even though this book targets Python 2, I have also provided a Python 3 version of every program listing for which the appropriate libraries are available. This means that you can take the techniques you learn in these pages and transfer them to the new version of the language by simply comparing the printed listing with the Python 3 version that you download.

There are two command-line prompts used in the book, and they are used consistently in their respective contexts. A single `$` is used as the system prompt at which the Python interpreter might be run, while the famous triple chevron `>>>` is used for interactive Python interpreter sessions themselves.

If you are familiar with good Python coding practices and with PEP-8, the community's coding standard, you will note that the program listings here in the printed book deviate in a number of respects from best practices. You can find PEP-8 here: `www.python.org/dev/peps/pep-0008/`.

I have varied from standard coding style in the interest of saving trees and to adapt the code to the printed page. For example, I often shamelessly import several modules in a single statement, instead of putting each module on its own line. My listings also tend to run commands without performing the familiar check for whether the script has really been run from the command line:

```
if __name__ == '__main__':
»   ...
```

This, again, is simply in the interest of space and readability. In the versions of the program listings provided in the downloadable source code bundle, I have tried to adopt a coding style closer to PEP-8, so do not be surprised if those listings look a bit different from the ones you see here in the book.

Your Comments

If you need to contact me directly about anything in the book, my e-mail address is `brandon@rhodesmill.org`, and I welcome ideas, questions, or constructive criticism. But you should submit any errata directly to the Apress web site on the page for this particular book, so that they can maintain the central list of what will have to be revised for the next printing.

Be well; speak respectfully to everyone; write careful tests; and use your newfound powers for good.Audience

CHAPTER 1

Introduction to Client/Server Networking

This book is about network programming with the Python language: about accomplishing a specific set of tasks that all involve a particular technology—computer networks—using a general-purpose programming language that can do all sorts of things besides the things that you will see illustrated in this book.

We lack the space between the covers of this book to teach you how to program in Python if you have never seen the language before, or never even written a computer program at all. So this book presumes that you have already learned something about Python programming from the many excellent tutorials and books on the subject. We hope that the Python examples in the book are good ones, from which you can learn how to structure and write your own Python programs. But we will be using all sorts of advanced Python features without explanation or apology—though, occasionally, we might point out how we are using a particular technique or construction when we think it is particularly interesting or clever.

On the other hand, this book does *not* start by assuming that you know any networking! As long as you have ever used a web browser or sent an e-mail, you should know enough to start reading this book at the beginning and learn about computer networking along the way. We will approach networking from the point of view of an application programmer who is either implementing a network-connected service—like a web site, an email server, or a networked computer game—or else writing a client program that is designed to use such a service.

Note that you will not, however, learn how to set up or configure networks from this book, for the simple reason that the Python language is not usually involved when network engineers or system administrators sit down to build and configure their networks. Instead, computer networks are typically assembled from network switches, Ethernet cables, fiber optic strands, and painstakingly configured routers. You will have to learn about devices like those from a book that focuses on creating computer networks in the first place; this book instead will talk about writing programs that use a computer network once it is already set up and running.

The Building Blocks: Stacks and Libraries

As we begin to explore Python network programming, there are two concepts that will appear over and over again:

- The idea of a *protocol stack*, in which very simple network services are used as a foundation on which to build more sophisticated services.

- The fact that you will often be using Python *libraries* of prepared code—whether from the built-in standard library that ships with Python, or from third-party modules that you download and install—that already know how to speak the network protocol you want to use.

In many cases, network programming simply involves selecting and using a library that already supports the network operations you need to perform. A major purpose of this book is to introduce you to all of the key networking libraries available for Python, and to teach you about the lower-level network services on which those libraries are built—both so that you understand how the libraries work, and so that you will understand what is happening when something at a lower level goes wrong.

Let's begin with a very simple example. I have here a mailing address, which looks like this:

```
207 N. Defiance St
Archbold, OH
```

And I am interested in knowing the latitude and longitude of this physical address. It just so happens that Google provides a "Maps API" that can perform such a conversion. What would I have to do to take advantage of this network service from Python?

When looking at a new network service that you want to use, it is always worthwhile to start by finding out whether someone has already implemented the protocol—in this case, the Google Maps protocol—that your program will need to speak. Start by scrolling through the Python Standard Library documentation, looking for anything having to do with Google Maps:

```
http://docs.python.org/library/
```

Do you see anything? No, neither do I. But it is important for a Python programmer to look through the Standard Library's table of contents pretty frequently, even if you usually do not find what you are looking for, because each reading will make you more familiar with the services that do come included with Python.

Since the Standard Library does not have a package to help us, we can turn to the Python Package Index, an excellent resource for finding all sorts of general-purpose Python packages contributed by other programmers and organizations from across the world. You can also, of course, check the web site of the vendor whose service you will be using to see whether they provide a python library to access it. Or you can do a general Google search for "Python" plus the name of whatever web service you want to use, and see whether any of the first few results link to a package that you might want to try.

In this case, I searched the Python Package Index, which lives at this URL:

```
http://pypi.python.org/
```

There, I did a search for Google maps, and immediately found a package that is actually named `googlemaps` and that provides a clean interface to its features (though, you will note from its description, it is *not* vendor-provided, but was instead written by someone besides Google):

```
http://pypi.python.org/pypi/googlemaps/
```

This is such a common situation—that you find a Python package that sounds like it might already do exactly what you want, and that you want to try it out on your system—that we should pause for a moment and introduce you to the very best Python technology for quickly trying out a new library: virtualenv!

In the old days, installing a Python package was a gruesome and irreversible act that required administrative privileges on your machine and left your system Python install permanently altered. After several months of heavy Python development, your system Python install could become a wasteland of dozens of packages, all installed by hand, and you could even find that the new packages you tried to install would break because they were incompatible with one of the old packages sitting on your hard drive from a project that ended months ago.

Careful Python programmers do not suffer from this situation any longer. Many of us install only one Python package system-wide: virtualenv. Once virtualenv is installed, you have the power to create any number of small, self-contained "virtual Python environments" where packages can be installed, un-installed, and experimented with without contaminating your system-wide Python. When a particular project or experiment is over, you simply remove its virtual environment directory, and your system is clean.

In this case, we want to create a virtual environment in which to test the googlemaps package. If you have never installed virtualenv on your system before, visit this URL to download and install it:

http://pypi.python.org/pypi/virtualenv

Once you have virtualenv installed, you can create a new environment like this (on Windows, the directory containing the Python binary in the virtual environment will be named "Scripts" instead):

```
$ virtualenv --no-site-packages gmapenv
$ cd gmapenv
$ ls
bin/  include/  lib/
$ . bin/activate
$ python -c 'import googlemaps'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named googlemaps
```

As you can see, the googlemaps package is not yet available! To install it, use the pip command that is inside your virtualenv and that is now on your path thanks to the activate command that you ran:

```
$ pip install googlemaps
Downloading/unpacking googlemaps
  Downloading googlemaps-1.0.2.tar.gz (60Kb): 60Kb downloaded
  Running setup.py egg_info for package googlemaps
Installing collected packages: googlemaps
  Running setup.py install for googlemaps
Successfully installed googlemaps
Cleaning up...
```

The python binary inside the virtualenv will now have the googlemaps package available:

```
$ python -c 'import googlemaps'
```

Now that you have the googlemaps package installed, you should be able to run the simple program named search1.py.

Listing 1–1. *Fetching a Longitude and Latitude*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 1 - search1.py

from googlemaps import GoogleMaps
address = '207 N. Defiance St, Archbold, OH'
print GoogleMaps().address_to_latlng(address)
```

Running it at the command line, you should see a result like this:

```
$ python search1.py
(41.5228242, -84.3063479)
```

And there, right on your computer screen, is the answer to our question about the address's latitude and longitude! The answer has been pulled directly from Google web service. Our first example program is a rousing success.

Are you annoyed to have opened a book on Python network programming, only to have found yourself immediately directed to download and install an obscure package that turned what might have been an interesting network program into a boring three-line Python script? Be at peace! Ninety percent of the time, you will find that this is exactly how programming problems are solved: by finding other programmers in the Python community that have already tackled the problem you are facing, and building intelligently and briefly upon their solutions.

But, we are not yet done exploring our example. You have seen that a complex network service can often be accessed quite trivially. But what is behind the pretty `googlemaps` interface? How does the service actually work? We will now explore, in detail, how the sophisticated Google Maps service is actually just the top layer of a network stack that involves at least a half-dozen different levels.

Application Layers

Our first program listing used a third-party Python library, downloaded from the Python Package Index, to solve our problem. What if that library did not exist? What if we had to build a client for Google's Maps API on our own? For the answer, take a look at `search2.py`.

Listing 1–2. *Fetching a JSON Document from the Google Maps URL*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 1 - search2.py

import urllib, urllib2
try:
»   import json
except ImportError:  # for Python 2.5
»   import simplejson as json

params = {'q': '207 N. Defiance St, Archbold, OH',
»   »     'output': 'json', 'oe': 'utf8'}
url = 'http://maps.google.com/maps/geo?' + urllib.urlencode(params)

rawreply = urllib2.urlopen(url).read()
reply = json.loads(rawreply)
print reply['Placemark'][0]['Point']['coordinates'][:-1]
```

Running this Python program returns an answer quite similar to that of our first script:

```
$ python search2.py
[-84.3063479, 41.5228242]
```

Well, okay, the output is not *exactly* the same—we can see, for example, that the JSON protocol does not distinguish between a tuple and a list, and also that Google sends back the longitude and latitude in the opposite order from the one that the `googlemaps` module liked to expose. But, it is clear that this script has accomplished much the same thing as the first one.

In `search2.py`, we have stepped one rung down the ladder, and instead of using any third-party packages at all, we are calling routines from Python's built-in Standard Library. This code, it happens, will work only on Python 2.6 or above unless you use `pip` to install the third-party `simplejson` package.

The first thing that you will notice about this code is that the semantics offered by the higher-level `googlemaps` module are absent. Unless you look very closely at this code, you might not even see that it's

asking about a mailing address at all! Whereas `search1.py` asked directly for an address to be turned into a latitude and longitude, the second listing painstakingly builds a URL from separate query parameters whose purpose might not even be clear to you unless you have already read the Google Maps documentation. If you want to read their documentation, by the way, you can find the Google Maps API described here:

`http://code.google.com/apis/maps/documentation/geocoding/`

If you look closely at the dictionary of query parameters in `search2.py`, you will see that the Q parameter, as is usual for Google services, provides the query string that we are asking about. The other parameters indicate the format in which we want the output returned. When we receive a document back as a result of looking up this URL, we then have to manually interpret it as a JSON data structure and then look at the correct element inside it to find the latitude and longitude.

The `search2.py` script, then, does exactly the same thing as the first one—but instead of doing so in the language of addresses and latitudes, it talks about the gritty details of constructing a URL, and the document that is fetched by making a web request to retrieve that URL. This is a common difference when you step down a level from one layer of a network stack to the layer beneath it: whereas the upper layer talked about what a request *meant*, the lower level can see only the details of how the request is *constructed*.

Speaking a Protocol

So our second example script creates a URL and fetches the document that corresponds to it. That operation sounds quite simple, and, of course, your web browser works very hard to make it look quite elementary. But the real reason that a URL can be used to fetch a document, of course, is that the URL is a kind of a recipe that describes where to find—and how to fetch—a given document on the web. The URL consists of the name of a protocol, followed by the name of the machine where the document lives, and finishes with the path that names document on that machine. The reason, then, that the `search2.py` Python program is able to resolve the URL and fetch the document at all is that the URL provides instructions that tell a lower-level protocol how to find the document.

That lower-level protocol the URL uses, in fact, is the famous Hypertext Transfer Protocol, or HTTP, which is the basis of nearly all modern web communications. We will learn more about it in Section 2 of this book. It is HTTP that provides the mechanism by which Python's built-in `urllib` is able to fetch the result from Google Maps. What, do you think, would it look like if we were to strip that layer of magic off—what if we wanted to use HTTP to directly fetch the result? The result is shown in `search3.py`.

Listing 1–3. *Making a Raw HTTP Connection to Google Maps*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 1 - search3.py

import httplib
try:
»   import json
except ImportError:  # for Python 2.5
»   import simplejson as json

path = ('/maps/geo?q=207+N.+Defiance+St%2C+Archbold%2C+OH'
»   »   '&output=json&oe=utf8')

connection = httplib.HTTPConnection('maps.google.com')
connection.request('GET', path)
rawreply = connection.getresponse().read()
```

```
reply = json.loads(rawreply)
print reply['Placemark'][0]['Point']['coordinates'][:-1]
```

In this listing, all references to the idea of a URL have disappeared—in fact, none of Python's URL-related libraries are imported at all! Instead, we are here directly manipulating the HTTP protocol: asking it to connect to a specific machine, to issue a `GET` request with a path that we have constructed, and finally to read the reply directly from the HTTP connection. Instead of being able to conveniently provide our query parameters as separate keys-and-values in a dictionary, we are having to embed them directly, by hand, in the path that we are requesting by first writing a question mark (`?`) followed by the parameters in the format `name=value` and all separated by `&` characters.

The result of running the program, however, is much the same as for the programs shown previously:

```
$ python search3.py
[-84.3063479, 41.5228242]
```

As we will see throughout this book, HTTP is just one of many protocols for which the Python Standard Library provides a built-in implementation. In `search3.py`, instead of having to worry about all of the details of how HTTP works, our code can simply ask for a request to be sent and then take a look at the resulting response. The protocol details that the script has to deal with are, of course, more primitive than those of `search2.py`, because we have stepped down another level in the protocol stack, but at least we are still able to rely on the Standard Library to handle the actual network data and make sure we get it right.

A Raw Network Conversation

But, of course, HTTP cannot simply send data between two machines using thin air. Instead, the HTTP protocol must operate by using some even simpler abstraction. In fact, it uses the capacity of modern operating systems to support a plain-text network conversation between two different programs across an IP network. The HTTP protocol, in other words, operates by dictating *exactly* what the text of the messages will look like that pass back and forth between two hosts implementing the protocol.

When we move beneath HTTP to look at what happens below it, we are dropping down to the very lowest level of the network stack that we can still access easily from Python. Take a careful look at `search4.py`. It makes exactly the same networking request to Google Maps as our previous three programs, but it does so by sending a raw text message across the Internet and receiving a bundle of text in return.

Listing 1–4. *Talking to Google Maps Through a Bare Socket*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 1 - search4.py

import socket
sock = socket.socket()
sock.connect(('maps.google.com', 80))
sock.sendall(
»   'GET /maps/geo?q=207+N.+Defiance+St%2C+Archbold%2C+OH'
»   '&output=json&oe=utf8&sensor=false HTTP/1.1\r\n'
»   'Host: maps.google.com:80\r\n'
»   'User-Agent: search4.py\r\n'
»   'Connection: close\r\n'
»   '\r\n')
rawreply = sock.recv(4096)
print rawreply
```

In moving from `search3.py` to `search4.py`, we have passed an important threshold. In every previous program listing, we were using a Python library—written in Python itself—that knows how to speak a complicated network protocol on our behalf. But here we have reached the bottom: we are calling the raw `socket()` function that is provided by the host operating system to support basic network communications on an IP network. We are, in other words, using the same mechanisms that a low-level system programmer would use in the C language when writing this exact same network operation.

We will learn more about "sockets" over the next few chapters. For now, you can notice in `search4.py` that raw network communication is a matter of sending and receiving *strings*. The request that we send is one string, and the reply—that, in this case, we simply print to the screen—is another large string. The HTTP request, whose text you can see inside the `sendall()` function, consists of the word GET—the name of the operation we want performed—followed by the path of the document we want fetched and the version of HTTP we support:

```
GET /maps/geo...sensor=false HTTP/1.1
```

Then there are a series of headers that each consist of a name, a colon, and a value, and finally a newline that ends the request.

The reply—which will print as the script's output if you run `search4.py`—is shown as Listing 1–5. I chose to simply print the reply to the screen in this example, rather than write the complex text-manipulation code that would be able to interpret the response, because I thought that simply reading the HTTP reply on your screen would give you a much better idea of what it looks like than if you had to decipher code designed to interpret it.

Listing 1–5. *The Output of Running* `search4.py`

```
HTTP/1.1 200 OK
Content-Type: text/javascript; charset=UTF-8
Vary: Accept-Language
Date: Wed, 21 Jul 2010 16:10:38 GMT
Server: mafe
Cache-Control: private, x-gzip-ok=""
X-XSS-Protection: 1; mode=block
Connection: close

{
  "name": "207 N. Defiance St, Archbold, OH",
  "Status": {
    "code": 200,
    "request": "geocode"
  },
  "Placemark": [ {
    ...
    "Point": {
      "coordinates": [ -84.3063479, 41.5228242, 0 ]
    }
  } ]
}
```

You can see that the HTTP reply is quite similar in structure to the HTTP request: It begins with a status line, which is followed by a number of headers describing the format of the result. After a blank line, the result itself is shown: a JavaScript data structure (a format known as JSON) that answers our query by describing the geographic location that the Google Maps search has returned.

All of these status lines and headers, of course, are exactly the sort of low-level details that Python's `httplib` was taking care of in the earlier listings. Here, we see what the communication looks like if that layer of software is stripped away.

Turtles All the Way Down

I hope you have enjoyed these initial examples of what Python network programming can look like. Stepping back, we can use this series of examples to make several points about network programming in Python.

First, you can perhaps now see more clearly what is meant by the term *protocol stack*: it means building a high-level, semantically sophisticated conversation—"I want the geographic location of this mailing address"—on top of simpler and more rudimentary conversations that ultimately are just sending text strings back and forth between two computers using their network hardware.

The protocol stack we have just explored, for example, is four protocols high:

- Google Maps URLs return JSON data containing coordinates.
- URLs name documents that can be retrieved using HTTP.
- HTTP uses sockets to support document commands like GET.
- Sockets know only how to send and receive text.

Each layer of the stack, you see, uses the tools provided by the layer beneath it, and in turn offers capabilities to the next higher layer.

A second point made clear through these examples is how very complete the Python support is for every one of the network levels at which we have just operated. Only when using a vendor-specific protocol, and needing to format requests so that Google would understand them, did we even have to resort to using a third-party library. Every single one of the other protocol levels we encountered already had strong support inside the Python Standard Library. Whether we wanted to fetch the document at a particular URL, or send and receive strings on a raw network socket, Python was ready with functions and classes that we could use to get the job done.

Third, note that my programs decreased considerably in quality as I forced myself to use increasingly lower-level protocols. The `search2.py` and `search3.py` listings, for example, started to hard-code things like the form structure and hostnames in a way that is very inflexible and might be rather hard to maintain later. The code in `search4.py` is even worse: it includes a handwritten, completely un-parameterized HTTP request whose structure is completely opaque to Python; and, of course, it contains none of the actual logic that would be necessary to parse and interpret the HTTP response and understand any network error conditions that might occur.

This illustrates a lesson that you should remember through every subsequent chapter of this book: that implementing network protocols correctly is difficult, and that you should use the Standard Library or third-party libraries whenever you possibly can. Especially when you are writing a network client, you will always find yourself tempted to oversimplify your code; you will tend to ignore many error conditions that might arise, to prepare for only the most likely responses, and, in general, to write very brittle code that knows as little about the service it is talking to as is technically possible. By instead using a third-party library that has developed a very thorough implementation of a protocol, because it has had to support many different Python developers who are using the library for a variety of tasks, you will benefit from all of the edge cases and awkward corners that the library implementer has already discovered and learned how to work around.

Fourth, it needs to be emphasized that higher-level network protocols—like the Google Maps protocol for resolving a street address—generally work by *hiding* the network layers beneath them. If you're using the `googlemaps` library, you might not even be aware that URLs and HTTP are the lower-level mechanisms that are being used to construct and answer your queries!

An interesting question, whose answer varies depending on how carefully a Python library has been written, is whether errors at those lower levels are correctly hidden by the library. Could a network error that makes Google temporarily unreachable from your site raise a raw, low-level networking exception in the middle of code that's just trying to find the coordinates of a street address? We will pay careful

attention to the topic of catching network errors as we go forward through this book, especially in the chapters of this first section, with their emphasis on low-level networking.

And for our final point, we reach the topic that will occupy us for the rest of this first section of the book: the fact that the `socket()` interface used in `search4.py` is *not*, in fact, the lowest protocol level in play when you make this request to Google! Just as our example has network protocols operating above the level above raw sockets, so also there are protocols down *beneath* the sockets abstraction that Python cannot see because your operating system manages them instead.

The layers operating below the `socket()` API are the following:

- The Transmission Control Protocol (TCP), which sockets use to support network conversations between two programs
- The Internet Protocol (IP), which knows how to send small messages call *packets* between different computers
- The “link layer,” at the very bottom, which consists of network hardware devices like Ethernet ports and wireless cards, which can send physical messages between directly-linked computers

Through the rest of this chapter, and in the two chapters that follow, we will explore these lowest protocol levels. We will start by examining the IP level, and then proceed in the following chapters to see how two quite different protocols—UDP and TCP—support the two basic kinds of conversation that are possible between applications on a pair of Internet-connected hosts.

The Internet Protocol

Both *networking*, which occurs when you connect several computers together so that they can communicate, and *internetworking*, which links adjacent networks together to form a much larger system like the Internet, are essentially just elaborate schemes to allow resource sharing.

All sorts of things in a computer, of course, need to be shared: disk drives, memory, and the CPU are all carefully guarded by the operating system so that the individual programs running on your computer can access those resources without stepping on each other’s toes. The network is yet another resource that the operating system needs to protect so that programs can communicate with one another without interfering with other conversations that happen to be occurring on the same network.

The physical networking devices that your computer uses to communicate—like Ethernet cards, wireless transmitters, and USB ports—are themselves each designed with an elaborate ability to share a single physical medium among many different devices that want to communicate. A dozen Ethernet cards might be plugged into the same hub; thirty wireless cards might be sharing the same radio channel; and a DSL modem uses frequency-domain multiplexing, a fundamental concept in electrical engineering, to keep its own digital signals from interfering with the analog signals sent down the line when you talk on the telephone.

The fundamental unit of sharing among network devices—the currency, if you will, in which they trade—is the “packet.” A packet is a binary string whose length might range from a few bytes to a few thousand bytes, which is transmitted as a single unit between network devices. Although there are some specialized networks today, especially in realms like telecommunications, where each individual byte coming down a transmission line might be separately routed to a different destination, the more general technologies used to build digital networks for modern computers are all based on the larger unit of the packet.

A packet often has only two properties at the physical level: the binary string that is the data it carries, and an address to which it is to be delivered. The address is usually a unique identifier that names one of the other network cards—and thus the computer behind it—attached to the same Ethernet segment or wireless channel as the computer transmitting the packet. The job of a network

card is to send and receive such packets without making the computer's operating system care about the details of how the network operates down at the level of wires and voltages.

What, then, is the Internet Protocol?

The Internet Protocol is a scheme for imposing a uniform system of addresses on all of the Internet-connected computers in the entire world, and to make it possible for packets to travel from one end of the Internet to the other. Ideally, an application like your web browser should be able to connect a host anywhere without ever knowing which maze of network devices each packet is traversing on its journey.

It is very rare for a Python program to operate at such a low level that it sees the Internet Protocol itself in action, but in many situations, it is helpful to at least know how it works.

IP Addresses

The Internet Protocol assigns a 4-byte address to every computer connected to the network. Such addresses are usually written as four decimal numbers, separated by periods, which each represent a single byte of the address. Each number can therefore range from 0 to 255. So an IP address looks like this:

```
130.207.244.244
```

Because purely numeric addresses can be difficult for humans to remember, the actual people using the Internet are generally shown *hostnames* rather than IP addresses. The user can simply type `google.com` and forget that behind the scene this resolves to an address like `74.125.67.103`, to which their computer can actually address packets for transmission over the Internet.

In `getname.py` you can see a very simple Python program that asks the operating system—Linux, Mac OS, Windows, or whatever system the program is running on—to resolve the hostname `google.com`. The particular network service, called the "Domain Name Service," that springs into action to answer hostname queries is fairly complex, and we will discuss it in greater detail in a subsequent chapter.

Listing 1–6. *Turning a Hostname into an IP Address*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 1 - getname.py

import socket
hostname = 'maps.google.com'
addr = socket.gethostbyname(hostname)
print 'The address of', hostname, 'is', addr
```

For now, you just need to remember two things:

- First, however fancy an Internet application might look, the actual Internet Protocol always uses 4-byte IP addresses to direct packets towards their destination.
- Second, the complicated details of how hostnames are resolved to IP addresses are usually handled by the operating system.

Like most details of the operation of the Internet Protocol, your operating system prefers to take care of them itself, hiding the details both from you and your Python code.

Actually, the addressing situation can be a bit more complex these days than the simple scheme just described. Because the world is beginning to run out of 4-byte IP addresses, an extended address scheme, called IPv6, is being deployed that allows absolutely gargantuan 16-byte addresses that should serve humanity's needs for a very long time to come. They are written differently from 4-byte IP addresses, and look like this:

```
fe80::fcfd:4aff:fecf:ea4e
```

But as long as your code accepts IP addresses or hostnames from the user and passes them directly to a networking library for processing, you will probably never need to worry about the distinction between IPv4 (the current version of the protocol) and IPv6. The operating system on which your Python code is running will know which IP version it is using and should interpret addresses accordingly.

Generally, IP addresses can be read from left to right: the first one or two bytes specify an organization, and then the next byte often specifies the particular subnet on which the target machine resides. The last byte narrows down the address to that specific machine or service. There are also a few special ranges of IP address that have a special meaning:

- *`127.*.*.*`*: IP addresses that begin with the byte `127` are in a special, reserved range that indicates they are local to the machine on which an application is running. When your web browser, or FTP client, or Python program connects to an address in this range, it is asking to speak to some other service or program that is running on the same machine. Most machines make use of only one address in this entire range: the IP address `127.0.0.1` is used universally to mean "this machine itself that this program is running on," and can often be accessed through the host name `localhost`.

- *`10.*.*.*`*, *`172.16–31.*.*`*, *`192.168.*.*`*: These IP ranges are reserved for what are called *private subnets*. The authorities who run the Internet have made an absolute promise: they will never hand out IP addresses in any of these three ranges to real companies setting up servers or services. Out on the Internet at large, therefore, these addresses are guaranteed to have no meaning; they name no host to which you could want to connect. Therefore, these addresses are free for you to use on any of your organization's internal networks where you want to be free to assign IP addresses internally, but whose hosts do not need to be accessible from other places on the Internet.

You are even likely to see some of these private addresses in your own home: your Linksys wireless router or DSL modem will often assign IP addresses from one of these private ranges to your home computers and laptops, and hide all of your Internet traffic behind the single "real" IP address that your Internet service provider has allocated for your use.

Routing

So, operating systems that implement the Internet protocol allow programs to send messages whose destinations IP addresses—say, `8.8.4.4`—and to deliver each packet, the operating system has to decide how to transmit it using one of the physical networks to which the machine is connected. This decision—the decision of where to send each packet, based on the IP address that is its destination—is called *routing*.

Most, or perhaps all, of the Python code you write during your career will be running on hosts out at the very edge of the Internet: not on gateway machines, that sit *between* different Internet subnets, but on hosts with a single network interface that connects them to the rest of the world. For such machines, routing becomes a quite simple decision:

- If the IP address looks like `127.*.*.*`, then the operating system knows that the packet is destined for another application running on the same machine.

- If the IP address is in the same subnet as the machine itself, then the destination host can be found by simply checking the local Ethernet segment, wireless channel, or whatever the local network happens to be, and sending the packet to a locally connected machine.
- Otherwise, your machine forwards the packet to a *gateway machine* that connects your local subnet to the rest of the Internet. It will then be up to the gateway machine to decide where to send the packet after that.

Of course, routing is only this simple at the very edge of the Internet, where the only decisions are whether to keep the packet on the local network or to send it winging its way across the rest of the Internet. You can imagine that routing decisions are much more complex for the dedicated network devices that form the Internet's backbone! There, on the hubs that connect entire continents, elaborate routing tables have to be constructed, consulted, and constantly updated in order to know that packets destined for Google go in one direction, but packets directed to a Yahoo IP address go in another, and that packets directed to your machine go in yet another. But it is very rare for Python applications to run on Internet backbone routers, so the simpler routing situation just outlined is nearly always the one you will see in action.

(That previous paragraph simplifies things a bit, of course; in reality, big service providers like Google and Yahoo have data centers all over the world, and might even lease space in some of the same facilities—so there is really no "one direction" in which to send Google packets and another direction to send Yahoo packets!)

I have been a bit vague in the previous paragraphs about how your computer decides whether an IP address belongs to a local subnet, or whether it should instead be forwarded through a gateway to the rest of the Internet. To illustrate the idea of a subnet, all of whose hosts share the same IP address prefix, I have been writing the prefix followed by asterisks for the parts of the address that could vary. Of course, the binary logic that runs your operating system's network stack does not actually insert little ASCII asterisks into its routing table! Instead, subnets are specified by combining an IP address with a mask that indicates how many of its most significant bits have to match to make a host belong to that subnet. If you keep in mind that every byte in an IP address represents eight bits of binary data, then you will be able to read subnet numbers very easily. They look like this:

- *`127.0.0.0/8`*: This pattern, which describes the IP address range that we discussed previously, which is reserved for the local host, specifies that the first eight bits (one byte) must match the number 127, and that the remaining 24 bits (three bytes) can have any value they want.
- *`192.168.0.0/16`*: This pattern will match any IP address that belongs in the private 192.168 range, because the first 16 bits must match perfectly. The last 16 bits of the 32-bit address are allowed to have whatever value they want to.
- *`192.168.5.0/24`*: Here we have a specification for one particular individual subnet. This is probably the most common kind of subnet mask on the entire Internet. The first three bytes of the address are completely specified, and have to match for an IP address to fall into this range. Only the very last byte (the last eight bits) is allowed to vary between machines in this range. This leaves 256 unique addresses. Typically, the `.0` address is used as the name of the subnet, and the `.255` address is used to create a "broadcast packet" that addresses all of the hosts on the subnet (as we will see in the next chapter), which leaves 254 addresses free to actually be assigned to computers. The address `.1` is very often used for the gateway that connects the subnet to the rest of the Internet, but some companies and schools choose to use another number for their gateways instead.

In nearly all cases, your Python code will simply rely on its host operating system to make packet routing choices correctly—just as it lets the operating system resolve host names to IP addresses in the first place.

Packet Fragmentation

One last Internet Protocol concept that deserves mention is packet fragmentation. While it is supposed to be a very obscure detail that is successfully hidden from your program by the cleverness of your operating system's network stack, it has caused enough problems over the Internet's history that it deserves at least a brief mention here.

Fragmentation is necessary because the Internet Protocol supports very large packets—they can be up to 64 kB in length—but the actual network devices from which IP networks are built usually support much smaller packet sizes. Ethernet networks, for example, support only 1,500-byte packets. Internet packets therefore include a "don't fragment" (DF) flag with which the sender can choose what they want to happen if the packet proves too small to fit across one of the physical networks that lies between the source computer and the destination:

- If the DF flag is unset, then fragmentation is permitted, and when the packet reaches the threshold of the network onto which it cannot fit, the gateway can split it into smaller packets and mark them to be reassembled at the other end.
- If the DF flag is set, then fragmentation is prohibited, and if the packet cannot fit, then it will be discarded and an error message will be sent back—in a special signaling packet called an "Internet Control Message Protocol" (ICMP) packet—to the machine that sent the packet so that it can try splitting the message into smaller pieces and re-sending it.

Your Python programs will usually have no control over the DF flag; instead, it is set by the operating system. Roughly, the logic that the system will usually use is this: if you are having a UDP conversation (see Chapter 2) that consists of individual datagrams winging their way across the Internet, then the operating system will leave DF unset so that each datagram reaches the destination in however many pieces are needed; but if you are having a TCP conversation (see Chapter 3) whose long stream of data might be hundreds or thousands of packets long, then the operating system will set the DF flag so that it can choose exactly the right packet size to let the conversation flow smoothly, without its packets constantly being fragmented *en route*, which makes the conversation slightly less efficient.

The biggest packet that an Internet subnet can accept is called its "maximum transmission unit" (MTU), and there used to be a big problem with MTU processing that caused problems for lots of Internet users. Back in the 1990s, Internet service providers (most notably phone companies offering DSL links) started using PPPoE, a protocol that puts IP packets inside a capsule that leaves them room for only 1,492 bytes instead of the full 1,500 bytes usually permitted across Ethernet. Many Internet sites were unprepared for this, because they used 1,500-byte packets by default and had blocked all ICMP packets as a misguided security measure. As a consequence, their servers could never receive the ICMP errors telling them that their large, "Don't Fragment" packets were reaching customers' DSL links and were unable to fit across them.

The maddening symptom of this situation was that small files or web pages could be viewed without a problem, and interactive protocols like Telnet and SSH would work since both of these activities send small packets anyway. But once the customer tried downloading a large file, or once a Telnet or SSH command resulted in several screens full of output at once, the connection would freeze and become unresponsive.

Today this problem is only very rarely encountered, but it illustrates how a low-level IP feature can generate a user-visible symptoms—and, therefore, why it is good to keep all of the features of IP in mind when writing and debugging network programs.

Learning More About IP

In the next chapters, we will step up to the protocol layers above IP and see how your Python programs can have different kinds of network conversations by using the different services built on top of the Internet Protocol. But, what if you have been intrigued by the preceding outline of how IP works, and want to learn more?

The official resources that describe the Internet Protocol are the "Requests for Comment" (RFCs) published by the IETF that describe exactly how the protocol works. They are carefully written and, when combined with a strong cup of coffee and a few hours of free reading time, will let you in on every single detail of how the Internet Protocols operate. Here, for example, is the RFC that defines the Internet Protocol itself:

`http://tools.ietf.org/html/rfc791`

You can also find RFCs referenced on general resources like Wikipedia, and RFCs will often cite other RFCs that describe further details of a protocol or addressing scheme.

If you want to learn everything about the Internet Protocol and the other protocols that run on top of it, you might be interested in acquiring the venerable text *TCP/IP Illustrated, Vol. 1: The Protocols*, by W. Richard Stevens. It covers, in very fine detail, all of the protocol operations at which this book will only have the space to gesture. There are also other good books on networking in general, and that might help with network configuration in particular if setting up IP networks and routing is something you do either at work or even just at home to get your computers on the Internet.

CHAPTER 2

UDP

The previous chapter asserted that all network communications these days are built atop the transmission of short messages called *packets* that are usually no longer than a few thousand bytes. Packets each wing their way across the network independently, free to take different paths toward the same destination if redundant or load-balanced routers are part of the network. This means that packets can arrive out of order. If network conditions are poor, or a packet is simply unlucky, then it might easily not arrive at all.

When a network application is built on top of IP, its designers face a fundamental question: will the network conversations in which the application will engage best be constructed from individual, unordered, and unreliable network packages? Or will their application be simpler and easier to write if the network instead appears to offer an ordered and reliable stream of bytes, so that their clients and servers can converse as though talking to a local pipe?

There are three possible approaches to building atop IP. Here they are, in order of decreasing popularity!

- The vast majority of applications today are built atop TCP, the Transmission Control Protocol, which offers ordered and reliable data streams between IP applications. We will explore its possibilities in Chapter 3.
- A few protocols, usually with short, self-contained requests and responses, and simple clients that will not be annoyed if a request gets lost and they have to repeat it, choose UDP, the User Datagram Protocol, described in this chapter.
- Very specialized protocols avoid both of these options, and choose to create an entirely new IP-based protocol that sits alongside TCP and UDP as an entirely new way of having conversations across an IP network.

The last of these three options is very rare. Normal operating system users are usually not even *allowed* to communicate on the network without going through TCP or UDP, which is how UDP gets its name: it is the way that normal "Users," as opposed to operating system administrators, can send packet-based messages.

While writing raw network packets is useful for network discovery programs like `ping` and `nmap`, this is a very specialized use case, and this book will not discuss how to build and transmit raw packets using Python. If you need this capability, find some example C code for constructing the packets that you need to forge, and try making the same low-level calls to `socket()` from Python.

So that leaves us with the normal, user-accessible IP protocols, TCP and UDP. We are covering UDP first in this book because even though it is used far less often than TCP, its simplicity will give us a window onto how network packets actually behave, which will be helpful when we then examine how TCP works.

Another reason for making UDP the subject of this second chapter is that while it can be more complicated to use than TCP—after all, it does so little for you, and you have to remember to watch for dropped or re-ordered packets yourself—its programming interface is correspondingly simpler, and will

give us good practice with the Python network API before we move on to the additional complications that are brought by the use of TCP.

Should You Read This Chapter?

Yes, you should read this chapter—and the next one on TCP—if you are going to be doing any programming on an IP network. The issues raised and answered are simply too fundamental. A good understanding of what is happening down at these low levels will serve you very well, regardless of whether you are fetching pages from a web server, or sending complicated queries to an industrial database.

But should you use what you learn in this chapter? Probably not! Unless you are talking to a service that already speaks UDP because of someone else's decision, you will probably want to use something else. The days when it was useful to sit down with a UDP connection and bang out packets toward another machine are very nearly gone.

The deployment of UDP is even rather dangerous for the general health of the IP network. The sophisticated TCP protocol will automatically back off as the network becomes saturated and starts to drop packets. But few UDP programmers want to even think about the complexity of typical congestion-avoidance algorithms—much less implement them correctly—with the result that a naively-written application atop UDP can bring a network to its knees, flooding your bandwidth with an increasing number of re-tries until almost no requests are actually getting through successfully.

If you even think you want to use the UDP protocol, then you probably want to use a message queue system instead. Take a look at Chapter 8, and you will probably find that ØMQ lets you do everything you wanted to accomplish with UDP, while having been programmed by people who dove far deeper into the efficiencies and quirks of the typical operating system network stack than you could do without months of research. If you need persistence or a broker, then try one of the message queues that come with their own servers for moving messages between parts of your application.

Use UDP only if you really want to be interacting with a very low level of the IP network stack. But, again, be sure to read this whole chapter either way, so that you know the details of what lies beneath some of your favorite protocols like DNS, real-time audio and video chat, and DHCP.

Addresses and Port Numbers

The IP protocol that we learned about in Chapter 1 assigns an IP address—which traditionally takes the form of a four-octet code, like `18.9.22.69`—to every machine connected to an IP network. In fact, it does a bit more than this: a machine with several network cards connected to the network will typically have a different IP address for each card, so that other hosts can choose the network over which you want to contact the machine. Multiple interfaces are also used to improve redundancy and bandwidth.

But even if an IP-connected machine has only one network card, we learned that it also has at least one other network address: the address `127.0.0.1` is how machines can connect to themselves. It serves as a stable name that each machine has for itself, that stays the same as network cables are plugged and unplugged and as wireless signals come and go.

And these IP addresses allow millions of different machines, using all sorts of different network hardware, to pass packets to each other over the fabric of an IP network.

But with UDP and TCP we now take a big step, and stop thinking about the routing needs of the network as a whole and start considering the needs of specific applications that are running on a particular machine. And the first thing we notice is that a single computer today can have many dozens of programs running on it at any given time—and many of these will want to use the network at the same moment! You might be checking e-mail with Thunderbird while a web page is downloading in Google Chrome, or installing a Python package with `pip` over the network while checking the status of a remote

server with SSH. Somehow, all of those different and simultaneous conversations need to take place without interfering with each other.

This is a general problem in both computer networking and electromagnetic signal theory. It is known as the need for *multiplexing*: the need for a single channel to be shared unambiguously by several different conversations. It was famously discovered that radio signals can be separated from one another by using different frequencies. To distinguish among the different destinations to which a UDP packet might be addressed—where all we have to work with are alphabets of symbols—the designers of IP chose the rough-and-ready technique of labeling each UDP packet with an unsigned 16-bit number (which therefore has a range of 0 to 65,536) that identifies a *port* to which an application can be attached and listening.

Imagine, for example, that you set up a DNS server (Chapter 4) on one of your machines, with the IP address 192.168.1.9. To allow other computers to find the service, the server will ask the operating system for permission to take control of the UDP port with the standard DNS port number 53. Assuming that no process is already running that has claimed that port number, the DNS server will be granted that port.

Next, imagine that a client machine with the IP address 192.168.1.30 on your network is given the IP address of this new DNS server and wants to issue a query. It will craft a DNS query in memory, and then ask the operating system to send that block of data as a UDP packet. Since there will need to be some way to identify the client when the packet returns, and since the client has not explicitly requested a port number, the operating system assigns it a random one—say, port 44137.

The packet will therefore wing its way toward port 53 with labels that identify its source as the IP address and UDP port numbers (here separated by a colon):

```
192.168.1.30:44137
```

And it will give its destination as the following:

```
192.168.1.9:53
```

This destination address, simple though it looks—just the number of a computer, and the number of a port—is everything that an IP network stack needs to guide this packet to its destination. The DNS server will receive the request from its operating system, along with the originating IP and port number. Once it has formulated a response, the DNS server will ask the operating system to send the response as a UDP packet to the IP address and UDP port number from which the request originally came.

The reply packet will have the source and destination swapped from what they were in the original packet, and upon its arrival at the source machine, it will be delivered to the waiting client program.

Port Number Ranges

So the UDP scheme is really quite simple; an IP address and port are all that is necessary to direct a packet to its destination.

As you saw in the story told in the previous section, if two programs are going to talk using UDP, then one of them has to send the first packet. Unavoidably, this means that the first program to talk—which is generally called the *client*—has to somehow know the IP address and port number that it should be sending that first packet to. The other program, the *server* who can just sit and wait for the incoming connection, does not necessarily need prior knowledge of the client because it can just read client IP addresses and port numbers off of the request packets as they first arrive.

The terms *client* and *server* generally imply a pattern where the server runs at a known address and port for long periods of time, and may answer millions of requests from thousands of other machines. When this pattern does not pertain—when two programs are not in the relationship of a client demanding a service and a busy server providing it—then you will often see programs cooperating with sockets called *peers* of each other instead.

How do clients learn the IP addresses and ports to which they should connect? There are generally three ways:

- *Convention*: Many port numbers have been designated as the official, well-known ports for specific services by the IANA, the Internet Assigned Numbers Authority. That is why we expected DNS to run at UDP port 53 in the foregoing example.
- *Automatic configuration*: Often the IP addresses of critical services like DNS are learned when a computer first connects to a network, if a protocol like DHCP is used. By combining these IP addresses with well-known port numbers, programs can reach these essential services.
- *Manual configuration*: For all of the situations that are not covered by the previous two cases, some other scheme will have to deliver an IP address or the corresponding hostname.

There are all kinds of ways that IP addresses and port numbers can be provided manually: asking a user to type a hostname; reading one from a configuration file; or learning the address from another service. There was, once, even a movement afoot to popularize a `portmap` daemon on Unix machines that would always live at port 2049 and answer questions about what ports *other* running programs were listening on!

When making decisions about defining port numbers, like 53 for the DNS, the IANA thinks of them as falling into three ranges—and this applies to both UDP and TCP port numbers:

- “Well-Known Ports” (0–1023) are for the most important and widely-used protocols. On many Unix-like operating systems, normal user programs cannot use these ports, which prevented troublesome undergraduates on multi-user machines from running programs to masquerade as important system services. Today the same protections apply when hosting companies hand out command-line Linux accounts.
- “Registered Ports” (1024–49151) are not usually treated as special by operating systems—any user can write a program that grabs port 5432 and pretends to be a PostgreSQL database, for example—but they can be registered by the IANA for specific protocols, and the IANA recommends that you avoid using them for anything but their assigned protocol.
- The remaining port numbers (49152–65535) are free for any use. They, as we shall see, are the pool on which modern operating systems draw in order to generate random port numbers when a client does not care what port it is assigned.

When you craft programs that accept port numbers from user input like the command line or configuration files, it is friendly to allow not just numeric port numbers but to let users type human-readable names for well-known ports. These names are standard, and are available through the `getservbyname()` call supported by Python’s standard `socket` module. If we want to ask where the Domain Name Service lives, we could have found out this way:

```
>>> import socket
>>> socket.getservbyname('domain')
53
```

As we will see in Chapter 4, port names can also be decoded by the more complicated `getaddrinfo()` function, which also lives in the `socket` module.

The database of well-known service names is usually kept in the file `/etc/services` on Unix machines, which you can peruse at your leisure. The lower end of the file, in particular, is littered with ancient protocols that still have reserved numbers despite not having had an actual packet addressed to

them anywhere in the world for many years. An up-to-date (and typically *much* more extensive) copy is also maintained online by the IANA at `/www.iana.org/assignments/port-numbers`.

The foregoing discussion, as we will learn in Chapter 3, applies equally well to TCP communications, and, in fact, the IANA seems to consider the port-number range to be a single resource shared by both TCP and UDP. They never assign a given port number to one service under TCP but to another service under UDP, and, in fact, usually assign both the UDP and TCP port numbers to a given service even if it is very unlikely to ever use anything other than TCP.

Sockets

Enough explanation! It is time to show you source code.

Rather than trying to invent its own API for doing networking, Python made an interesting decision: it simply provides a slightly object-based interface to all of the normal, gritty, low-level operating system calls that are normally used to accomplish networking tasks on POSIX-compliant operating systems.

This might look like laziness, but it was actually brilliance, and for two different reasons! First, it is very rare for programming language designers, whose expertise lies in a different area, to create a true improvement over an existing networking API that—whatever its faults—was created by actual network programmers. Second, an attractive object-oriented interface works well until you need some odd combination of actions or options that was perfectly well-supported by grungy low-level operating system calls, but that seems frustratingly impossible through a prettier interface.

In fact, this was one of the reasons that Python came as such a breath of fresh air to those of us toiling in lower-level languages in the early 1990s. Finally, a higher-level language had arrived that let us make low-level operating system calls when we needed them without insisting that we try going through an awkward but ostensibly “prettier” interface first!

So, Python exposes the normal POSIX calls for raw UDP and TCP connections rather than trying to invent any of its own. And the normal POSIX networking calls operate around a central concept called a *socket*.

If you have ever worked with POSIX before, you will probably have run across the fact that instead of making you repeat a file name over and over again, the calls let you use the file name to create a “file descriptor” that represents a connection to the file, and through which you can access the file until you are done working with it.

Sockets provide the same idea for the networking realm: when you ask for access to a line of communication—like a UDP port, as we are about to see—you create one of these abstract “socket” objects and then ask for it to be bound to the port you want to use. If the binding is successful, then the socket “holds on to” that port number for you, and keeps it in your possession until such time as you “close” the socket to release its resources.

In fact, sockets and file descriptors are not merely similar concepts; sockets actually *are* file descriptors, which happen to be connected to network sources of data rather than to data stored on a filesystem. This gives them some unusual abilities relative to normal files. But POSIX also lets you perform normal file operations on them like `read()` and `write()`, meaning that a program that just wants to read or write simple data can treat a socket as though it were a file without knowing the difference!

What do sockets look like in operation? Take a look at Listing 2–1, which shows a simple server and client. You can see already that all sorts of operations are taking place that are drawn from the `socket` module in the Python Standard Library.

Listing 2–1. *UDP Server and Client on the Loopback Interface*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 2 - udp_local.py
# UDP client and server on localhost

import socket, sys
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

MAX = 65535
PORT = 1060

if sys.argv[1:] == ['server']:
»   s.bind(('127.0.0.1', PORT))
»   print 'Listening at', s.getsockname()
»   while True:
»   »   data, address = s.recvfrom(MAX)
»   »   print 'The client at', address, 'says', repr(data)
»   »   s.sendto('Your data was %d bytes' % len(data), address)

elif sys.argv[1:] == ['client']:
»   print 'Address before sending:', s.getsockname()
»   s.sendto('This is my message', ('127.0.0.1', PORT))
»   print 'Address after sending', s.getsockname()
»   data, address = s.recvfrom(MAX)  # overly promiscuous - see text!
»   print 'The server', address, 'says', repr(data)

else:
»   print >>sys.stderr, 'usage: udp_local.py server|client'
```

You should be able to run this script right on your own computer, even if you are not currently in the range of a network, because both server and client use only the "localhost" IP address. Try running the server first:

```
$ python udp_local.py server
Listening at ('127.0.0.1', 1060)
```

After printing this line of output, the server hangs and waits for an incoming message. In the source code, you can see that it took three steps for the server to get up and running.

It first created a plain socket with the `socket()` call. This new socket has no name, is not yet connected to anything, and will raise an exception if you attempt any communications with it. But the socket is, at least, marked as being of a particular type: its family is `AF_INET`, the Internet family of protocols, and it is of the `SOCK_DGRAM` datagram type, which means UDP. (The term "datagram" is the official term for an application-level block of transmitted data. Some people call UDP packets "datagrams"—like Candygrams, I suppose, but with data in them instead.)

Next, this simple server uses the `bind()` command to request a UDP network address, which you can see is a simple tuple containing an IP address (a hostname is also acceptable) and a UDP port number. At this point, an exception could be raised if another program is already using that UDP port and the server script cannot obtain it. Try running another copy of the server—you will see that it complains:

```
$ python udp_local.py server
Traceback (most recent call last):
  ...
socket.error: [Errno 98] Address already in use
```

Of course, there is some very small chance that you got this error the *first* time you ran the server, because port 1060 was already in use on your machine. It happens that I found myself in a bit of a bind when choosing the port number for this first example. It had to be above 1023, of course, or you could not have run the script without being a system administrator—and, while I really do like my little example scripts, I really do not want to encourage anyone running them as the system administrator! I could have let the operating system choose the port number (as I did for the client, as we will see in a moment) and had the server print it out and then made you type it into the client as one of its

command-line arguments, but then I would not have gotten to show you the syntax for asking for a particular port number yourself. Finally, I considered using a port from the high-numbered "ephemeral" range previously described, but those are precisely the ports that might randomly already be in use by some other application on your machine, like your web browser or SSH client.

So my only option seemed to be a port from the reserved-but-not-well-known range above 1023. I glanced over the list and made the gamble that you, gentle reader, are not running SAP BusinessObjects Polestar on the laptop or desktop or server where you are running my Python scripts. If you are, then try changing the `PORT` constant in the script to something else, and you have my apologies.

Note that the Python program can always use a socket's `getsockname()` method to retrieve the current IP and port to which the socket is bound.

Once the socker has been bound successfully, the server is ready to start receiving requests! It enters a loop and repeatedly runs `recvfrom()`, telling the routine that it will happily receive messages up to a maximum length of `MAX`, which is equal to 65535 bytes—a value that happens to be the greatest length that a UDP packet can possibly have, so that we will always be shown the full content of each packet. Until we send a message with a client, our `recvfrom()` call will wait forever.

So let's start up our client and see the result. The client code is also shown in Listing 2–1, beneath the test of `sys.argv` for the string `'client'`.

(I hope, by the way, that it is not confusing that this example—like some of the others in the book—combines the server and client code into a single listing, selected by command-line arguments; I often prefer this style since it keeps server and client logic close to each other on the page, and makes it easier to see which snippets of server code go with which snippets of client code.)

While the server is still running, open another command window on your system, and try running the client twice in a row like this:

```
$ python udp_local.py client
Address before sending: ('0.0.0.0', 0)
Address after sending ('0.0.0.0', 33578)
The server ('127.0.0.1', 1060) says 'Your data was 18 bytes'
$ python udp_local.py client
Address before sending: ('0.0.0.0', 0)
Address after sending ('0.0.0.0', 56305)
The server ('127.0.0.1', 1060) says 'Your data was 18 bytes'
```

Over in the server's command window, you should see it reporting each connection that it serves:

```
The client at ('127.0.0.1', 41201) says, 'This is my message'
The client at ('127.0.0.1', 59490) says, 'This is my message'
```

Although the client code is slightly simpler than that of the server—there are only two substantial lines of code—it introduces several new concepts.

First, the client takes the time to attempt a `getsockname()` before any address has been assigned to the socket. This lets us see that both IP address and port number start as all zeroes—a new socket is a blank slate. Then the client calls `sendto()` with both a message and a destination address; this simple call is all that is necessary to send a packet winging its way toward the server! But, of course, we need an IP address and port number ourselves, on the client end, if we are going to be communicating. So the operating system assigns one automatically, as you can see from the output of the second call to `getsockname()`. And, as promised, the client port numbers are each from the IANA range for "ephemeral" port numbers (at least they are here, on my laptop, under Linux; under a different operating system, you might get different results).

Since the client knows that he is expecting a reply from the server, he simply calls the socket's `recv()` method without bothering with the `recvfrom()` version that also returns an address. As you can see from their output, both the client and the server are successfully seeing each other's messages; each time the client runs, a complete round-trip of request and reply is passing between two UDP sockets. Success!

Unreliability, Backoff, Blocking, Timeouts

Because the client and server in the previous section were both running on the same machine and talking through its loopback interface—which is not even a physical network card that could experience a signaling glitch and lose a packet, but merely a virtual connection back to the same machine deep in the network stack—there was no real way that packets could get lost, and so we did not actually see any of the inconvenience of UDP. How does code change when packets could really be lost?

Take a look at Listing 2–2. Unlike the previous example, you can run this client and server on two different machines on the Internet. And instead of always answering client requests, this server randomly chooses to answer only half of the requests coming in from clients—which will let us demonstrate how to build reliability into our client code, without waiting what might be hours for a real dropped packet to occur!

Listing 2–2. *UDP Server and Client on Different Machines*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 2 - udp_remote.py
# UDP client and server for talking over the network

import random, socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

MAX = 65535
PORT = 1060

if 2 <= len(sys.argv) <= 3 and sys.argv[1] == 'server':
    interface = sys.argv[2] if len(sys.argv) > 2 else ''
    s.bind((interface, PORT))
    print 'Listening at', s.getsockname()
    while True:
        data, address = s.recvfrom(MAX)
        if random.randint(0, 1):
            print 'The client at', address, 'says:', repr(data)
            s.sendto('Your data was %d bytes' % len(data), address)
        else:
            print 'Pretending to drop packet from', address

elif len(sys.argv) == 3 and sys.argv[1] == 'client':
    hostname = sys.argv[2]
    s.connect((hostname, PORT))
    print 'Client socket name is', s.getsockname()
    delay = 0.1
    while True:
        s.send('This is another message')
        print 'Waiting up to', delay, 'seconds for a reply'
        s.settimeout(delay)
        try:
            data = s.recv(MAX)
        except socket.timeout:
            delay *= 2  # wait even longer for the next request
            if delay > 2.0:
```

```
»   »   »   »   raise RuntimeError('I think the server is down')
»   »   except:
»   »   »   raise   # a real error, so we let the user see it
»   »   else:
»   »   »   break   # we are done, and can stop looping

»   print 'The server says', repr(data)

else:
»   print >>sys.stderr, 'usage: udp_remote.py server [ <interface> ]'
»   print >>sys.stderr, '   or: udp_remote.py client <host>'
»   sys.exit(2)
```

While the server in our earlier example told the operating system that it wanted only packets that arrived from other processes on the same machine through the private `127.0.0.1` interface, this server is being more generous and inviting packets that arrive at the server through any network interface whatsoever. That is why we are specifying the server IP address as `''`, which means "any local interface," which my Linux laptop is translating to `0.0.0.0`, as we can see from the line that it prints out when it starts:

```
$ python udp_remote.py server
Listening at ('0.0.0.0', 1060)
```

As you can see, each time a request is received, the server uses `randint()` to flip a coin to decide whether this request will be answered, so that we do not have to keep running the client all day waiting for a real dropped packet. Whichever decision it makes, it prints out a message to the screen so that we can keep up with its activity.

So how do we write a "real" UDP client, one that has to deal with the fact that packets might be lost?

First, UDP's unreliability means that the client has to perform its request inside a loop, and that it, in fact, has to be somewhat arbitrary—actually, quite aggressively arbitrary—in deciding when it has waited "too long" for a reply and needs to send another one. This difficult choice is necessary because there is generally no way for the client to distinguish between three quite different events:

- The reply is taking a long time to come back, but will soon arrive.
- The reply will never arrive because it, or the request, was lost.
- The server is down and is not replying to anyone.

So a UDP client has to choose a schedule on which it will send duplicate requests if it waits a reasonable period of time without getting a response. Of course, it might wind up wasting the server's time by doing this, because the first reply might be about to arrive and the second copy of the request might cause the server to perform needless duplicate work. But at some point the client must decide to re-send, or it risks waiting forever.

So rather than letting the operating system leave it forever paused in the `recv()` call, this client first does a `settimeout()` on the socket. This informs the system that the client is unwilling to stay stuck waiting inside a socket operation for more than `delay` seconds, and wants the call interrupted with a `socket.timeout` exception once a call has waited for that long.

A call that waits for a network operation to complete, by the way, is said to "block" the caller, and the term "blocking" is used to describe a call like `recv()` that can make the client wait until new data arrives. When we get to Chapter 6 and discuss server architecture, the distinction between blocking and non-blocking network calls will loom very large!

This particular client starts with a modest tenth-of-a-second wait. For my home network, where ping times are usually a few dozen milliseconds, this will rarely cause the client to send a duplicate request simply because the reply is delayed in getting back.

A very important feature of this client is what happens if the timeout is reached. It does *not* simply start sending out repeat requests over and over again at a fixed interval! Since the leading cause of packet loss is congestion—as anyone knows who has tried sending normal data upstream over a DSL modem at the same time as photographs or videos are uploading—the last thing we want to do is to respond to a possibly dropped packet by sending even more of them.

Therefore, this client uses a technique known as *exponential backoff*, where its attempts become less and less frequent. This serves the important purpose of surviving a few dropped requests or replies, while making it possible that a congested network will slowly recover as all of the active clients back off on their demands and gradually send fewer packets. Although there exist fancier algorithms for exponential backoff—for example, the Ethernet version of the algorithm adds some randomness so that two competing network cards are unlikely to back off on exactly the same schedule—the basic effect can be achieved quite simply by doubling the delay each time that a reply is not received.

Please note that if the requests are being made to a server that is 200 milliseconds away, this naive algorithm will *always* send at least two packets because it will never learn that requests to this server always take more than 0.1 seconds! If you are writing a UDP client that lives a long time, think about having it save the value of `delay` between one call and the next, and use this to adjust its expectations so that it gradually comes to accept that the server really is 200 milliseconds away and that the network is not simply always dropping the first request!

Of course, you do not want to make your client become intolerably slow simply because one request ran into trouble and ran the `delay` up very high. A good technique might be to set a timer and measure how long the successful calls to the server take, and use this to adjust `delay` back downward over time once a string of successful requests has taken place. Something like a moving average might be helpful.

When you run the client, give it the hostname of the other machine on which you are running the server script, as shown previously. Sometimes, this client will get lucky and get an immediate reply:

```
$ python udp_remote.py client guinness
Client socket name is ('127.0.0.1', 45420)
Waiting up to 0.1 seconds for a reply
The server says 'Your data was 23 bytes'
```

But often it will find that one or more of its requests never result in replies, and will have to re-try. If you watch its repeated attempts carefully, you can even see the exponential backoff happening in real time, as the `print` statements that echo to the screen come more and more slowly as the `delay` timer ramps up:

```
$ python udp_remote.py client guinness
Client socket name is ('127.0.0.1', 58414)
Waiting up to 0.1 seconds for a reply
Waiting up to 0.2 seconds for a reply
Waiting up to 0.4 seconds for a reply
Waiting up to 0.8 seconds for a reply
The server says 'Your data was 23 bytes'
```

You can see over at the server whether the requests are actually making it, or whether by any chance you hit a real packet drop on your network. When I ran the foregoing test, I could look over at the server's console and see that all of the packets had actually made it:

```
Pretending to drop packet from ('192.168.5.10', 53322)
Pretending to drop packet from ('192.168.5.10', 53322)
Pretending to drop packet from ('192.168.5.10', 53322)
Pretending to drop packet from ('192.168.5.10', 53322)
The client at ('192.168.5.10', 53322) says, 'This is another message'
```

What if the server is down entirely? Unfortunately, UDP gives us no way to distinguish between a server that is down and a network that is simply in such poor condition that it is dropping all of our packets. Of course, I suppose we should not blame UDP for this problem; the fact is, simply, that the

world itself gives us no way to distinguish between something that we cannot detect and something that does not exist! So the best that the client can do is give up once it has made enough attempts. Kill the server process, and try running the client again:

```
$ python udp_remote.py client guinness
Waiting up to 0.1 seconds for a reply
Waiting up to 0.2 seconds for a reply
Waiting up to 0.4 seconds for a reply
Waiting up to 0.8 seconds for a reply
Waiting up to 1.6 seconds for a reply
Traceback (most recent call last):
  ...
RuntimeError: I think the server is down
```

Of course, giving up makes sense only if your program is trying to perform some brief task and needs to produce output or return some kind of result to the user. If you are writing a daemon program that runs all day—like, say, a weather icon in the corner of the screen that displays the temperature and forecast fetched from a remote UDP service—then it is fine to have code that keeps re-trying "forever." After all, the desktop or laptop machine might be off the network for long periods of time, and your code might have to patiently wait for hours or days until the forecast server can be contacted again.

If you are writing daemon code that re-tries all day, then do *not* adhere to a strict exponential backoff, or you will soon have ramped the delay up to a value like two hours, and then you will probably miss the entire half-hour period during which the laptop owner sits down in a coffee shop and you could actually have gotten to the network! Instead, choose some maximum delay—like, say, five minutes—and once the exponential backoff has reached that period, keep it there, so that you are always guaranteed to attempt an update once the user has been on the network for five minutes after a long time disconnected.

Of course, if your operating system lets your process be signaled for events like the network coming back up, then you will be able to do much better than to play with timers and guess about when the network might come back! But system-specific mechanisms like that are, sadly, beyond the scope of this book, so let's now return to UDP and a few more issues that it raises.

Connecting UDP Sockets

Listing 2–2, which we examined in the previous section, introduced another new concept that needs explanation. We have already discussed binding—both the explicit `bind()` call that the server uses to grab the port number that it wants to use, as well as the implicit binding that takes place when the client first tries to use the socket and is assigned a random ephemeral port number by the operating system.

But this remote UDP client also uses a new call that we have not discussed before: the `connect()` socket operation. You can see easily enough what it does. Instead of having to use `sendto()` and an explicit UDP address every time we want to send something to the server, the `connect()` call lets the operating system know ahead of time which remote address to which we want to send packets, so that we can simply supply data to the `send()` call and not have to repeat the server address again.

But `connect()` does something else important, which will not be obvious at all from reading the Listing 2–2 script.

To approach this topic, let us return to Listing 2–1 for a moment. You will recall that both its client and server use the loopback IP address and assume reliable delivery—the client will wait forever for a response. Try running the client from Listing 2–1 in one window:

```
$ python udp_local.py client
Address before sending: ('0.0.0.0', 0)
Address after sending ('0.0.0.0', 47873)
```

The client is now waiting—perhaps forever—for a response in reply to the packet it has just sent to the localhost IP address at UDP port 1060. But what if we nefariously try sending it back a packet from a *different* server, instead?

From another command prompt on the same system, try running Python and entering these commands—and for the port number, copy the integer that was just printed to the screen when you ran the UDP client:

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> s.sendto('Fake reply', ('127.0.0.1', 47873))
10
```

(You can see, by the way, that the actual return value of `sendto()` is the length of the UDP packet that was sent.)

Even though this Python session has grabbed a completely wild port number, which looks nothing like 1060 nor is even close, the client happily accepts this fake reply as an answer and prints it to the screen! Returning to the command line where the client was running, we see the following:

```
The server ('127.0.0.1', 49371) says 'Fake reply'
$
```

Disaster! It turns out that our first client accepts answers from *anywhere*. Even though the server is running on the localhost, and remote network connectivity is not even desirable for Listing 1-1, the client will even accept packets from another machine! If I bring up a Python prompt on another box and run the same two lines of code as just shown, then a waiting client can even see the remote IP address:

```
The server ('192.168.5.10', 59970) says 'Fake reply from elsewhere'
$
```

If a real UDP client were written this way, and an attacker or malcontent knew that we were running it, then they could send packets to its UDP port—or, even if they did not know its address, they could quickly flood random ports on your machine, hoping to find the client—and feed it a different answer than the server would have given it.

Now you can return to Listing 2–2, and if you will perform the foregoing tests, you will find that this second client is *not* susceptible to receiving packets from other servers. This is because of the second, less-obvious effect of using `connect()` to select a UDP socket's destination instead of specifying the address each time yourself: once you have run `connect()`, the operating system will discard any incoming packets to your port whose return address and port number do not match the server to which you are sending packets.

There are, then, two ways to write UDP clients that are careful about the return addresses of the packets arriving back:

- You can use `sendto()` and direct each outgoing packet to a specific destination, and then use `recvfrom()` to receive the replies and carefully check the return address it gives you against the list of servers to which you have made outstanding requests.
- You can `connect()` your socket right after creating it, and then simply use `send()` and `recv()`, and the operating system will filter out unwanted packets for you. This works only for speaking to one server at a time, because running `connect()` a second time on the same socket does not add a second destination address to your UDP socket. Instead, it wipes out the first address entirely, so that no further replies from the earlier address will be delivered to your program.

After you have connected a UDP socket using `connect()`, you can use the socket's `getpeername()` method to remember the address to which you have connected it. Be careful about calling this on a

socket that is not yet connected, however; rather than returning `0.0.0.0` or some other wildcard response, the call will raise `socket.error` instead.

Two last points should be made about the `connect()` call.

First, doing a `connect()` on a UDP socket, of type `SOCK_DGRAM`, does *not* send any information across the network, nor do anything to warn the server that packets might be coming. It simply writes the address and port number into the operating system's memory for use when you run `send()` and `recv()`, and then returns control to your program without doing any actual network communication.

Second, doing a `connect()`, or even filtering out unwanted packets yourself using the return address, is *not* a form of security! If there is someone on the network who is really malicious, it is usually easy enough for their computer to forge packets with the server's return address so that their faked replies will make it in past your address filter just fine.

Sending packets with another computer's return address is called *spoofing*, and is one of the first things that protocol designers have to think about when designing protocols that are supposed to be safe against interference. See Chapter 6 for more information.

Request IDs: A Good Idea

The messages sent in both Listings 2–1 and 2–2 were simple text. But if you should ever design a scheme of your own for doing UDP requests and responses, you should strongly consider adding a sequence number to each request and making sure that the reply you accept uses the same number. On the server side, you will just copy the number from each request into the reply that the server sends back. This has at least two big advantages.

First, it protects you from being confused by duplicate answers to requests that you repeated several times in your exponential backoff loop. You can see easily enough how this could happen: you send request A; you get bored waiting for an answer; so you repeat request A and then you finally get an answer, reply A. You assume that the first copy got lost, so you continue merrily on your way.

But—what if *both* requests made it to the server, and the replies have been just a bit slow in making it back, and so you have received one of the two replies but the other is about to arrive? If you now send request B to the server and start listening, you will almost immediately receive the duplicate reply A, and perhaps think that it is the answer to the question you asked in request B and become very confused. You could from then on wind up completely out of step, interpreting each reply as corresponding to a different request than the one that you think it does!

Request IDs protect you against that. If you gave every copy of request A the request ID #42496, and request B the ID #16916, then the program loop waiting for the answer to B can simply keep throwing out replies whose IDs do not equal #16916 until it finally receives one that matches. This protects against duplicate replies, which arise not only in the case where you repeated the question, but also in rare circumstances because a redundancy in the network fabric accidentally generates two copies of the packet somewhere between the server and the client.

The other purpose that request IDs can serve is to provide a barrier against spoofing, at least in the case where the attackers cannot see your packets. If they can, of course, then you are completely lost: they will see the IP, port number, and request ID of every single packet you send, and can try sending fake replies—hoping that their answers arrive before those of the server, of course!—to any request that they like. But in the case where the attackers *cannot* observe your traffic, but have to shoot UDP packets at your server blindly, a good-sized request ID number can make it much less likely that their answer will not be discarded by your client.

You will note that the example request IDs that I used in the story I just told were neither sequential, nor easy to guess—precisely so that an attacker would have no idea what a likely sequence number is. If you start with 0 or 1 and count upward from there, you make an attacker's job much easier. Instead, try using the `random` module to generate large integers. If your ID number is a random number between 0 and N, then an attacker's chance of hitting you with a valid packet—even assuming that the attacker knows the server's address and port—is at most $1/N$, and maybe much less if he or she has to wildly try hitting all possible port numbers on your machine.

But, of course, none of this is real security—it just protects against naive spoofing attacks from people who cannot observe your network traffic. Real security protects you even if attackers can both observe your traffic and insert their own messages whenever they like. In Chapter 6, we will look at how real security works.

Binding to Interfaces

So far we have seen two possibilities for the IP address used in the `bind()` call that the server makes: you can use `'127.0.0.1'` to indicate that you only want packets from other programs running on the same machine, or use an empty string `''` as a wildcard, indicating that you are willing to receive packets from any interface.

It actually turns out that there is a third choice: you can provide the IP address of one of the machine's external IP interfaces, like its Ethernet connection or wireless card, and the server will listen only for packets destined for those IPs. You might have noticed that Listing 2–2 actually allows us to provide a server string for the `bind()` call, which will now let us do a few experiments.

First, what if we bind solely to an external interface? Run the server like this, using whatever your operating system tells you is the external IP address of your system:

```
$ python udp_remote.py server 192.168.5.130
Listening at ('192.168.5.130', 1060)
```

Connecting to this IP address from another machine should still work just fine:

```
$ python udp_remote.py client guinness
Client socket name is ('192.168.5.10', 35084)
Waiting up to 0.1 seconds for a reply
The server says 'Your data was 23 bytes'
```

But if you try connecting to the service through the loopback interface by running the client script on the *same* machine, the packets will never be delivered:

```
$ python udp_remote.py client 127.0.0.1
Client socket name is ('127.0.0.1', 60251)
Waiting up to 0.1 seconds for a reply
Traceback (most recent call last):
  ...
socket.error: [Errno 111] Connection refused
```

Actually, on my operating system at least, the result is even better than the packets never being delivered: because the operating system can see whether one of its own ports is opened without sending a packet across the network, it immediately replies that a connection to that port is impossible!

You might think this means that programs running on the localhost cannot now connect to the server. Unfortunately, you would be wrong! Try running the client again on the same machine, but this time use the external IP address of the box, even though the client and server are both running there:

```
$ python udp_remote.py client 192.168.5.130
Client socket name is ('192.168.5.130', 34919)
Waiting up to 0.1 seconds for a reply
The server says 'Your data was 23 bytes'
```

Do you see what happened? Programs running locally are allowed to send requests that originate from any of the machine's IP addresses that they want—even if they are just using that IP address to talk back to another service on the same machine!

So binding to an IP interface might limit which external hosts can talk to you; but it will certainly not limit conversations with other clients on the same machine, so long as they know the IP address that they should use to connect.

Second, what happens if we try to run *two* servers at the same time? Stop all of the scripts that are running, and we can try running two servers on the same box. One will be connected to the loopback:

```
$ python udp_remote.py server 127.0.0.1
Listening at ('127.0.0.1', 1060)
```

And then we try running a second one, connected to the wildcard IP address that allows requests from any address:

```
$ python udp_remote.py server
Traceback (most recent call last):
  ...
socket.error: [Errno 98] Address already in use
```

Whoops! What happened? We have learned something about operating system IP stacks and the rules that they follow: they do not allow two different sockets to listen at the same IP address and port number, because then the operating system would not know where to deliver incoming packets. And both of the servers just shown wanted to hear packets coming from the localhost to port 1060.

But what if instead of trying to run the second server against all IP interfaces, we just ran it against an external IP interface—one that the first copy of the server is *not* listening to? Let us try:

```
$ python udp_remote.py server 192.168.5.130
Listening at ('192.168.5.130', 1060)
```

It worked! There are now two servers running on this machine, one of which is bound to the inward-looking port 1060 on the loopback interface, and the other looking outward for packets arriving on port 1060 from the network to which my wireless card has connected. If you happen to be on a box with several remote interfaces, you can start up even more servers, one on each remote interface.

Once you have these servers running, try to send them some packets with the client program. You will find that each request is received by only one server, and that in each case it will be the server that holds the particular IP address to which you have directed the UDP request packet.

The lesson of all of this is that an IP network stack never thinks of a UDP port as a lone entity that is either entirely available, or else in use, at any given moment. Instead, it thinks in terms of UDP "socket names" that are always a pair linking an IP interface—even if it is the wildcard interface—with a UDP port number. It is these socket names that must not conflict among the listening servers at any given moment, rather than the bare UDP ports that are in use.

One last warning: since the foregoing discussion indicated that binding your server to the interface `127.0.0.1` protects you from possibly malicious packets generated on the external network, you might think that binding to one external interface will protect you from malicious packets generated by malcontents on other external networks. For example, on a large server with multiple network cards, you might be tempted to bind to a private subnet that faces your other servers, and think thereby that you will avoid spoofed packets arriving at your Internet-facing public IP address.

Sadly, life is not so simple. It actually depends on your choice of operating system, and then upon how it is specifically configured, whether inbound packets addressed to one interface are allowed to appear at another interface. It might be that your system will quite happily accept packets that claim to be from other servers on your network if they appear over on your public Internet connection! Check with your operating system documentation, or your system administrator, to find out more about your particular case. Configuring and running a firewall on your box could also provide protection if your operating system does not.

UDP Fragmentation

I have been claiming so far in this chapter that UDP lets you, as a user, send raw network packets to which just a little bit of information (an IP address and port for both the sender and receiver) has been added. But you might already have become suspicious, because the foregoing program listings have suggested that a UDP packet can be up to 64kB in size, whereas you probably already know that your Ethernet or wireless card can only handle packets of around 1,500 bytes instead.

The actual truth is that IP sends small UDP packets as single packets on the wire, but splits up larger UDP packets into several small physical packets, as was briefly discussed in Chapter 1. This means that large packets are more likely to be dropped, since if any one of their pieces fails to make its way to the destination, then the whole packet can never be reassembled and delivered to the listening operating system.

But aside from the higher chance of failure, this process of fragmenting large UDP packets so that they will fit on the wire should be invisible to your application. There are three ways, however, in which it might be relevant:

- If you are thinking about efficiency, you might want to limit your protocol to small packets, to make retransmission less likely and to limit how long it takes the remote IP stack to reassemble your UDP packet and give it to the waiting application.
- If the ICMP packets are wrongfully blocked by a firewall that would normally allow your host to auto-detect the MTU between you and the remote host, then your larger UDP packets might disappear into oblivion without your ever knowing. The MTU is the “maximum transmission unit” or “largest packet size” that all of the network devices between two hosts will support.
- If your protocol can make its own choices about how it splits up data between different packets, and you want to be able to auto-adjust this size based on the actual MTU between two hosts, then some operating systems let you turn off fragmentation and receive an error if a UDP packet is too big. This lets you regroup and split it into several packets if that is possible.

Linux is one operating system that supports this last option. Take a look at Listing 2–3, which sends a very large message to one of the servers that we have just designed.

Listing 2–3. *Sending a Very Large UDP Packet*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 2 - big_sender.py
# Send a big UDP packet to our server.

import IN, socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

MAX = 65535
PORT = 1060

if len(sys.argv) != 2:
»   print >>sys.stderr, 'usage: big_sender.py host'
»   sys.exit(2)

hostname = sys.argv[1]
```

```
s.connect((hostname, PORT))
s.setsockopt(socket.IPPROTO_IP, IN.IP_MTU_DISCOVER, IN.IP_PMTUDISC_DO)
try:
    s.send('#' * 65000)
except socket.error:
    print 'The message did not make it'
    option = getattr(IN, 'IP_MTU', 14)  # constant taken from <linux/in.h>
    print 'MTU:', s.getsockopt(socket.IPPROTO_IP, option)
else:
    print 'The big message was sent! Your network supports really big packets!'
```

If we run this program against a server elsewhere on my home network, then we discover that my wireless network allows physical packets that are no bigger than the 1,500 bytes typically supported by Ethernet-style networks:

```
$ python big_sender.py guinness
The message did not make it
MTU: 1500
```

It is slightly more surprising that the loopback interface on my laptop, which presumably could support packets as large as my RAM, also imposes an MTU that is far short of the maximum UDP packet length:

```
$ python big_sender.py localhost
The message did not make it
MTU: 16436
```

But, the ability to check the MTU is a fairly obscure feature. As you can see from the program listing, Python 2.6.5 on my machine for some reason fails to include the `IP_MTU` socket option name that is necessary to determine a socket's current MTU, so I had to manually copy the integer option code 14 out of one of the system C header files. So you should probably ignore the issue of fragmentation and, if you worry about it at all, try to keep your UDP packets short; but this example was at least useful in showing you that fragmentation does need to take place, in case you run into any of its consequences!

Socket Options

The POSIX socket interface also supports all sorts of *socket options* that control specific behaviors of network sockets. These are accessed through the Python socket methods `getsockopt()` and `setsockopt()`, using the options you will find documented for your operating system. On Linux, for example, try viewing the manual pages *socket(7)*, *udp(7)*, and—when you progress to the next chapter—*tcp(7)*.

When setting socket options, you have to first name the "option group" in which they live, and then as a subsequent argument name the actual option you want to set; consult your operating system manual for the names of these groups. See Listing 2–3 in the next section for some example real-world calls involving socket options. Just like the Python calls `getattr()` and `setattr()`, the set call simply takes one more argument:

```
value = s.getsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST)
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, value)
```

Many options are specific to particular operating systems, and may be finicky about how their options are presented. Here are some of the more common:

- `SO_BROADCAST`: Allows broadcast UDP packets to be sent and received; see the next section for details.

- SO_DONTROUTE: Only be willing to send packets that are addressed to hosts on subnets to which this computer is connected directly. My laptop, for example, at this moment would be willing to send packets to the networks 127.0.0.0/8 and 192.168.5.0/24 if this socket option were set, but would not be willing to send them anywhere else.
- SO_TYPE: When passed to getsockopt(), this returns to you regardless of whether a socket is of type SOCK_DGRAM and can be used for UDP, or it is of type SOCK_STREAM and instead supports the semantics of TCP (see Chapter 3).

The next chapter will introduce some further socket options that apply specifically to TCP sockets.

Broadcast

If UDP has a superpower, it is its ability to support broadcast: instead of sending a packet to some specific other host, you can point it at an entire subnet to which your machine is attached and have the physical network card broadcast the packet so that all attached hosts see it without its having to be copied separately to each one of them.

Now, it should be immediately mentioned that broadcast is considered passé these days, because a more sophisticated technique called “multicast” has been developed, which lets modern operating systems take better advantage of the intelligence built into many networks and network interface devices. Also, multicast can work with hosts that are not on the local subnet, which is what makes broadcast unusable for many applications! But if you want an easy way to keep something like gaming clients or automated scoreboards up-to-date on the local network, and each client can survive the occasional dropped packet, then UDP broadcast is an easy choice.

Listing 2–4 shows an example of a server that can receive broadcast packets and a client that can send them. And if you look closely, you will see that there is pretty much just one difference between this listing and the techniques we were using in previous listings: before using this socket object, we are using its setsockopt() method to turn on broadcast. Aside from that, the socket is used quite normally by both server and client.

Listing 2–4. *UDP Broadcast*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 2 - udp_broadcast.py
# UDP client and server for broadcast messages on a local LAN

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

MAX = 65535
PORT = 1060

if 2 <= len(sys.argv) <= 3 and sys.argv[1] == 'server':
    s.bind(('', PORT))
    print 'Listening for broadcasts at', s.getsockname()
    while True:
        data, address = s.recvfrom(MAX)
        print 'The client at %r says: %r' % (address, data)

elif len(sys.argv) == 3 and sys.argv[1] == 'client':
    network = sys.argv[2]
    s.sendto('Broadcast message!', (network, PORT))
```

```
else:
    print >>sys.stderr, 'usage: udp_broadcast.py server'
    print >>sys.stderr, '   or: udp_broadcast.py client <host>'
    sys.exit(2)
```

When trying this server and client out, the first thing you should notice is they behave exactly like a normal client and server if you simply use the client to send packets that are addressed to the IP address of a particular server. Turning on broadcast for a UDP socket does *not* disable or change its normal ability to send and receive specifically addressed packets.

The magic happens when you view the settings for your local network, and use its IP "broadcast address" as the destination for the client. First bring up one or two servers on your network, using commands like the following:

```
$ python udp_broadcast.py server
Listening for broadcasts at ('0.0.0.0', 1060)
```

Then, while they are running, first use the client to send messages to each server. You will see that only one server gets each message:

```
$ python udp_broadcast.py client 192.168.5.10
```

But when you use the local network's broadcast address, suddenly you will see that all of the broadcast servers get the packet at the same time! (But no normal servers will see it—run a few copies of the normal `udp_remote.py` server while making broadcasts to be convinced!) On my local network at the moment, the `ifconfig` command tells me that the broadcast address is this:

```
$ python udp_broadcast.py client 192.168.5.255
```

And, sure enough, both servers immediately report that they see the message! In case your operating system makes it difficult to determine the broadcast address, and you do not mind doing a broadcast out of every single network port of your host, Python lets you use the special hostname `'<broadcast>'` when sending with a UDP socket. Be careful to quote that name when passing it to our client, since the < and > characters are quite special to any normal POSIX shell:

```
$ python udp_broadcast.py client "<broadcast>"
```

If there were any platform-independent way to learn each connected subnet and its broadcast address, I would show you; but unfortunately you will have to consult your own operating system documentation if you want to do anything more specific than use this special `'<broadcast>'` string.

When to Use UDP

You might think that UDP would be very efficient for sending small messages. Actually, UDP is efficient only if your host ever only sends one message at a time, then waits for a response. If your application might send several messages in a burst, then using an intelligent message queue algorithm like ØMQ will actually be more efficient because it will set a short timer that lets it bundle several small messages together to send them over a single round-trip to the server, probably on a TCP connection that does a much better job of splitting the payload into fragments than you would!

There are two good reasons to use UDP:

- Because you are implementing a protocol that already exists, and it uses UDP
- Because unreliable subnet broadcast is a great pattern for your application, and UDP supports it perfectly

Outside of these two situations, you should probably look at later chapters of this book for inspiration about how to construct the communication for your application.

Summary

The User Data Protocol, UDP, lets user-level programs send individual packets across an IP network. Typically, a client program sends a packet to a server, which then replies back using the return address built into every UDP packet.

The POSIX network stack gives you access to UDP through the idea of a “socket,” which is a communications endpoint that can sit at an IP address and UDP port number—these two things together are called the socket’s “name”—and send and receive UDP packets. These primitive network operations are offered by Python through the built-in `socket` module.

The server needs to `bind()` to an address and port before it can receive incoming packets. Client UDP programs can just start sending, and the operating system will choose a port number for them automatically.

Since UDP is built atop the actual behavior of network packets, it is unreliable: packets can be dropped either because of a glitch on a network transmission medium, or because a network segment becomes too busy. Clients have to compensate for this by being willing to re-transmit a request until they receive a reply. To prevent making a busy network even worse, clients should use exponential backoff as they encounter repeated failure, and should also make their initial wait time longer if they find that round-trips to the server are simply taking longer than their author expected.

Request IDs are crucial to combat the problem of reply duplication, where a reply you thought was lost arrives later after all and could be mistaken for the reply to your current question. If randomly chosen, request IDs can also help protect against naive spoofing attacks.

When using sockets, it is important to distinguish the act of “binding”—by which you grab a particular UDP port for the use of a particular socket—from the act that the client performs by “connecting,” which limits all replies received so that they can come only from the particular server to which you want to talk.

Among the socket options available for UDP sockets, the most powerful is broadcast, which lets you send packets to every host on your subnet without having to send to each host individually. This can help when programming local LAN games or other cooperative computation, and is one of the few reasons that you would select UDP for new applications.

CHAPTER 3

TCP

The Transmission Control Protocol (TCP) is the workhorse of the Internet. First defined in 1974, it lets applications send one another streams of data that, if they arrive at all—that is, unless a connection dies because of a network problem—are guaranteed to arrive intact, in order, and without duplication.

Protocols that carry documents and files nearly always ride atop TCP, including HTTP and all the major ways of transmitting e-mail. It is also the foundation of choice for protocols that carry on long conversations between people or computers, like SSH and many popular chat protocols.

When the Internet was younger, it was sometimes possible to squeeze a little more performance out of a network by building your application atop UDP and choosing the size and timing of each individual packet yourself. But modern TCP implementations tend to be very smart, having benefited from more than 30 years of improvement, innovation, and research, and these days even very performance-critical applications like message queues (Chapter 8) often choose TCP as their medium.

How TCP Works

As we learned in Chapter 2, real networks are fickle things that sometimes drop the packets you transmit across them, occasionally create extra copies of a packet instead, and are also known to deliver packets out of order. With a bare-packet facility like UDP, your own application code has to worry about whether messages arrived, and have a plan for recovering if they did not. But with TCP, the packets themselves are hidden and your application can simply stream data toward its destination, confident that it will be re-transmitted until it finally arrives.

The classic definition of TCP is RFC 793 from 1981, though many subsequent RFCs have detailed extensions and improvements.

How does TCP provide a reliable connection? It starts by combining two mechanisms that we discussed in Chapter 2. There, we had to implement them ourselves because we were using UDP. But with TCP they come built in, and are performed by the operating system's network stack without your application even being involved.

First, every packet is given a sequence number, so that the system on the receiving end can put them back together in the right order, and so that it can notice missing packets in the sequence and ask that they be re-transmitted.

Instead of using sequential integers (1,2,...) to mark packets, TCP uses a counter that counts the number of bytes transmitted. So a 1,024-byte packet with a sequence number of 7,200 would be followed by a packet with a sequence number of 8,224. This means that a busy network stack does not have to remember how it broke a data stream up into packets; if asked for a re-transmission, it can break the stream up into packets some other way (which might let it fit more data into a packet if more bytes are now waiting for transmission), and the receiver can still put the packets back together.

The initial sequence number, in good TCP implementations, is chosen randomly so villains cannot assume that every connection starts at byte zero and easily craft forged packets by guessing how far a transmission that they want to interrupt has proceeded.

Rather than running very slowly in lock-step by needing every packet to be acknowledged before it sends the next one, TCP sends whole bursts of packets at a time before expecting a response. The amount of data that a sender is willing to have on the wire at any given moment is called the size of the TCP “window.”

The TCP implementation on the receiving end can regulate the window size of the transmitting end, and thus slow or pause the connection. This is called “flow control.” This lets it forbid the transmission of additional packets in cases where its input buffer is full and it would have to discard any more data if it were to arrive right now.

Finally, if TCP sees that packets are being dropped, it assumes that the network is becoming congested and stops sending as much data every second. This can be something of a disaster on wireless networks and other media where packets are sometimes simply lost because of noise. It can also ruin connections that are running fine until a router reboots and the endpoints cannot talk for, say, 20 seconds; by the time the network comes back up, the two TCP peers will have determined that the network is quite extraordinarily overloaded with traffic, and will for some time afterward refuse to send each other data at anything other than a trickle.

The protocol involves many other nuances and details beyond the behaviors just described, but hopefully this description gives you a good feel for how it will work—even though, you will remember, all your application will see is a stream of data, with the actual packets and sequence numbers cleverly hidden away by your operating system network stack.

When to Use TCP

If your network programs are at all like mine, then most of the network communications you perform from Python will use TCP. You might, in fact, spend an entire career without ever deliberately generating a UDP packet from your code. (Though, as we will see in Chapter 5, UDP is probably involved every time your program needs to use a DNS hostname!)

Because TCP has very nearly become a universal default when two programs need to communicate, we should look at a few instances in which its behavior is not optimal for certain kinds of data, in case an application you are writing ever falls into one of these categories.

First, TCP is unwieldy for protocols where clients want to send single, small requests to a server, and then are done and will not talk to it further. It takes three packets for two hosts to set up a TCP connection—the famous sequence of SYN, SYN-ACK, and ACK (which mean “I want to talk, here is the packet sequence number I will be starting with”; “okay, here’s mine”; “okay!”)—and then another three or four to shut the connection back down (either a quick FIN, FIN-ACK, ACK, or a slightly longer pair of separate FIN and ACK packets). That is six packets just to send a single request! Protocol designers quickly turn to UDP in such cases.

One question to ask, though, is whether a client might want to open a TCP connection and then use it over several minutes or hours to make many separate requests to the same server. Once the connection was going and the cost of the handshake had been paid, each actual request and response would only require a single packet in each direction—and they would benefit from all of TCP’s intelligence about re-transmitting, exponential backing off, and flow control.

Where UDP really shines, then, is where such a long-term relationship does *not* pertain between client and server, and especially where there are so many clients that a typical TCP implementation would run out of port numbers if it had to keep up with a separate data stream for each active client.

The second situation where TCP is inappropriate is when an application can do something much smarter than simply re-transmit data when a packet has been lost. Imagine an audio chat conversation, for example: if a second’s worth of data is lost because of a dropped packet, then it will do little good to simply re-send that same second of audio, over and over, until it finally arrives.

Instead, the client should just paper over that awkward second with whatever audio it can piece together from the packets that did arrive (a clever audio protocol will begin and end each packet with a bit of heavily-compressed audio from the preceding and following moments of time for exactly this

situation), and then keep going after the interruption as though it did not occur. This is impossible with TCP, and so UDP is often the foundation of live-streaming multimedia over the Internet.

What TCP Sockets Mean

As was the case with UDP in Chapter 2, TCP uses port numbers to distinguish different applications running at the same IP address, and follows exactly the same conventions regarding well-known and ephemeral port numbers. Re-read the section “Addresses and Port Numbers” if you want to review the details.

As we saw in the previous chapter, it takes only a single socket to speak UDP: a server can open a datagram port and then receive packets from thousands of different clients. While it is possible to `connect()` a datagram socket to a particular conversation partner so that you always `send()` to one address and only `recv()` packets sent back from that address, the idea of a connection is just a convenience. The effect of `connect()` is exactly the same as your application simply deciding to send to only one address with `sendto()` calls, and then ignoring responses from any but that same address.

But with a stateful stream protocol like TCP, the `connect()` call becomes the fundamental act upon which all other network communication hinges. It is, in fact, the moment when your operating system’s network stack kicks off the handshake protocol just described that—if successful—will make both ends of the TCP stream ready for use.

And this means that a TCP `connect()` can fail. The remote host might not answer; it might refuse the connection; or more obscure protocol errors might occur like the immediate receipt of a RST (“reset”) packet. Because a stream connection involves setting up a persistent connection between two hosts, the other host needs to be listening and ready to accept your connection.

On the “server side”—which, for the purpose of this chapter, is the conversation partner not doing the `connect()` call but receiving the SYN packet that it initiates—an incoming connection generates an even more momentous event, the creation of a new socket! This is because the standard POSIX interface to TCP actually involves two completely different kinds of sockets: “passive” listening sockets and active “connected” ones.

- A passive socket holds the “socket name”—the address and port number—at which the server is ready to receive connections. No data can ever be received or sent by this kind of port; it does not represent any actual network conversation. Instead, it is how the server alerts the operating system to its willingness to receive incoming connections in the first place.

- An active, connected socket is bound to one particular remote conversation partner, who has their own IP address and port number. It can be used only for talking back and forth with that partner, and can be read and written to without worrying about how the resulting data will be split up into packets—in many cases, a connected socket can be passed to another POSIX program that expects to read from a normal file, and the program will never even know that it is talking to the network!

Note that while a passive socket is made unique by the interface address and port number at which it is listening (so that no one else is allowed to grab that same address and port), there can be many active sockets that all share the same local socket name. A busy web server to which a thousand clients have all made HTTP connections, for example, will have a thousand active sockets all bound to its public IP address at port 80. What makes an active socket unique is, rather, the four-part coordinate:

```
(local_ip, local_port, remote_ip, remote_port)
```

It is this four-tuple by which the operating system names each active TCP connection, and incoming TCP packets are examined to see whether their source and destination address associate them with any of the currently active sockets on the system.

A Simple TCP Client and Server

Take a look at Listing 3–1. As I did in the last chapter, I have here combined what could have been two separate programs into a single listing, both so that they can share a bit of common code (you can see that both the client and server create their TCP socket in the same way), and so that the client and server code are directly adjacent here in the book and you can read them together more easily.

Listing 3–1. *Simple TCP Server and Client*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 3 - tcp_sixteen.py
# Simple TCP client and server that send and receive 16 octets

import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = sys.argv.pop() if len(sys.argv) == 3 else '127.0.0.1'
PORT = 1060

def recv_all(sock, length):
»   data = ''
»   while len(data) < length:
»   »   more = sock.recv(length - len(data))
»   »   if not more:
»   »   »   raise EOFError('socket closed %d bytes into a %d-byte message'
»   »   »                  % (len(data), length))
»   »   data += more
»   return data

if sys.argv[1:] == ['server']:
»   s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
»   s.bind((HOST, PORT))
»   s.listen(1)
»   while True:
»   »   print 'Listening at', s.getsockname()
»   »   sc, sockname = s.accept()
»   »   print 'We have accepted a connection from', sockname
»   »   print 'Socket connects', sc.getsockname(), 'and', sc.getpeername()
»   »   message = recv_all(sc, 16)
»   »   print 'The incoming sixteen-octet message says', repr(message)
»   »   sc.sendall('Farewell, client')
»   »   sc.close()
»   »   print 'Reply sent, socket closed'

elif sys.argv[1:] == ['client']:
»   s.connect((HOST, PORT))
»   print 'Client has been assigned socket name', s.getsockname()
»   s.sendall('Hi there, server')
```

```
»   reply = recv_all(s, 16)
»   print 'The server said', repr(reply)
»   s.close()

else:
»   print >>sys.stderr, 'usage: tcp_local.py server|client [host]'
```

In Chapter 2, we approached the subject of bind() quite carefully, since the address we provide as its argument makes a very important choice: it determines whether remote hosts can try connecting to our server, or whether our server is protected against outside connections and will be contacted only by other programs running on the same machine. So Chapter 2 started with safe program listings that used only the localhost's loopback interface, which always has the IP address 127.0.0.1, and then progressed to more dangerous listings that allowed hosts anywhere on the Internet to connect to our sample code.

Here, we have combined both possibilities into a single listing. By default, this server code makes the safe choice of binding to 127.0.0.1, but we can supply a command-line argument to bind to one of our machine's external IP addresses instead—or we can even supply a blank string to indicate that we will accept connections at any of our machine's IP addresses whatever. Again, review Chapter 2 if you want to remember all the rules, which apply equally to TCP and UDP connections and sockets.

Our choice of port number is also the same as the one we made for our UDP port in Chapter 2 and, again, the symmetry between TCP and UDP on the subject of port numbers is close enough that you can simply apply the reasoning we used there to understand why the same choice has been used here in this chapter.

So what are the differences between our earlier efforts with UDP, and this new client and server that are instead built atop TCP?

The client actually looks much the same. It creates a socket, runs connect() with the address of the server with which it wants to communicate, and then is free to send and receive data. But notice that there are several differences.

First, the TCP connect() call—as we discussed a moment ago—is not the innocuous bit of local socket configuration that it is in the case of UDP, where it merely sets a default address used with any subsequent send() calls, and places a filter on packets arriving at our socket. Here, connect() is a real live network operation that kicks off the three-way handshake between the client and server machine so that they are ready to communicate. This means that connect() can fail, as you can verify quite easily by executing this script when the server is not running:

```
$ python tcp_sixteen.py client
Traceback (most recent call last):
  File "tcp_sixteen.py", line 29, in <module>
    s.connect((HOST, PORT))
  File "<string>", line 1, in connect
socket.error: [Errno 111] Connection refused
```

Second, you will see that this TCP client is in one way much simpler than our UDP client, because it does not need to make any provision for missing data. Because of the assurances that TCP provides, it can send() data without checking whether the remote end receives it, and run recv() without having to consider the possibility of re-transmitting its request. The client can rest assured that the network stack will perform any necessary re-transmission to get its data through.

Third, there is a direction in which this program is actually more complicated than the equivalent UDP code—and this might surprise you, because with all of its guarantees it sounds like TCP streams would be uniformly simpler to program with than UDP datagrams. But precisely because TCP considers your outgoing and incoming data to be, simply, streams, with no beginning or end, it feels free to split them up into packets however it wants. And this means that send() and recv() mean different things than they meant before. In the case of UDP, they simply meant "send this data in a packet" or "receive a single data packet," and so each datagram was atomic: it either succeeded or failed as an entire unit. You

will, at the application level, never see UDP packets that are only half-sent or half-received; only fully intact datagrams are delivered to the application.

But TCP might split data into several pieces during transmission and then gradually reassemble it on the receiving end. Although this is vanishingly unlikely with the small sixteen-octet messages in Listing 3–1, our code still needs to be prepared for the possibility. What are the consequences of TCP streaming for both our `send()` and our `recv()` calls?

When we perform a TCP `send()`, our operating system's networking stack will face one of three situations:

- The data can be immediately accepted by the system, either because the network card is immediately free to transmit, or because the system has room to copy the data to a temporary outgoing buffer so that your program can continue running. In these cases, `send()` returns immediately, and it will return the length of your data string because the whole string was transmitted.

- Another possibility is that the network card is busy and that the outgoing data buffer for this socket is full and the system cannot—or will not—allocate any more space. In this case, the default behavior of `send()` is simply to block, pausing your program until the data can be accepted.

- There is a final, hybrid possibility: that the outgoing buffers are almost full, but not quite, and so *part* of the data you are trying to send can be immediately queued, but the rest will have to wait. In this case, `send()` completes immediately and returns the number of bytes accepted from the beginning of your data string, but leaves the rest of the data unprocessed.

Because of this last possibility, you cannot simply call `send()` on a stream socket without checking the return value. Instead, you have to put a `send()` call inside a loop like this one, that—in the case of a partial transmission—keeps trying to send the remaining data until the entire string has been sent:

```
bytes_sent = 0
while bytes_sent < len(message):
»   message_remaining = message[bytes_sent:]
»   bytes_sent += s.send(message_remaining)
```

Fortunately, Python does not force us to do this dance ourselves every time we have a block of data to send: the Standard Library socket implementation provides a friendly `sendall()` method, which Listing 3–1 uses instead. Not only is `sendall()` faster than doing it ourselves because it is implemented in C, but (for those readers who know what this means) it releases the Global Interpreter Lock during its loop so that other Python threads can run without contention until all of the data has been transmitted.

Unfortunately, no equivalent is provided for the `recv()` call, despite the fact that it might return only part of the data that is on the way from the client. Internally, the operating system implementation of `recv()` uses logic very close to that used when sending:

- If no data is available, then `recv()` blocks and your program pauses until data arrives.

- If plenty of data is available already in the incoming buffer, then you are given as many bytes as you asked `recv()` for.

- But if the buffer contains a bit of data, but not as much as you are asking for, then you are immediately returned what does happen to be there, even if it is not as much as you have asked for.

That is why our `recv()` call has to be inside a loop: the operating system has no way of knowing that this simple client and server are using fixed-width sixteen-octet messages, and so the system cannot

guess when the incoming data might finally add up to what your program will consider a complete message.

Why does the Python Standard Library include `sendall()` but no equivalent for the `recv()` method? Probably because fixed-length messages are so uncommon these days. Most protocols have far more complicated rules about how part of an incoming stream is delimited than a simple decision that "the message is always 16 bytes long."

So, in most real-world programs, the loop that runs `recv()` is actually much more complicated than the one in Listing 3–1, because it often has to read or process part of the message before it can guess how much more is coming. For example, an HTTP request often consists of headers, a blank line, and then however many further bytes of data were specified in the `Content-length` header. You would not know how many times to keep running `recv()` until you had at least received the headers and then parsed them to find out the content length!

One Socket per Conversation

Turning to the server code in Listing 3–1, we see a very different pattern than we have seen before—and it is a difference that hinges on the very meaning of a TCP stream socket. Recall from our foregoing discussion that there are two very different kinds of stream sockets: listening sockets, with which servers make a port available for incoming connections, and connected sockets, which represent the conversation that a server is having with a particular client.

In Listing 3–1, you can see how this distinction is carried through in actual server code. The link, which might strike you as odd at first, is that a listening socket actually produces new connected sockets as the return value that you get by listening! Follow the steps in the program listing to see the order in which the socket operations occur.

First, the server runs `bind()` to claim a particular port. Note that this does not yet decide whether the socket will be a client or server socket—that is, whether it will be actively making a connection or passively waiting to receive incoming connections. It simply claims a particular port, either on a particular interface or all interfaces, for the use of this socket. Clients can use this call if, for some reason, they want to reach out to a server from a particular port on their machine rather than simply accepting whatever ephemeral port number they would otherwise be assigned.

The real moment of decision comes with the next method call, when the server announces that it wants to use the socket to `listen()`. Running this on a TCP socket utterly transforms its character: after `listen()` has been called, the socket is irrevocably changed and can never, from this point on, be used to send or receive data. That particular socket object will now never be connected to any specific client. Instead, the socket can now be used only to receive incoming connections through its `accept()` method—a method that we have not seen yet in this book, because its purpose is solely to support listening TCP sockets—and each of these calls waits for a new client to connect and then returns an entirely new socket that governs the new conversation that has just started with them!

As you can see from the code, `getsockname()` works fine against both listening and connected sockets, and in both cases lets you find out what local TCP port the socket is occupying. To learn the address of the client to which a connected socket is connected, you can at any time run the `getpeername()` method, or you can store the socket name that is returned as the second return value from `accept()`. When we run this server, we see that both values give us the same address:

```
$ python tcp_sixteen.py server
Listening at ('127.0.0.1', 1060)
We have accepted a connection from ('127.0.0.1', 58185)
Socket connects ('127.0.0.1', 1060) and ('127.0.0.1', 58185)
The incoming sixteen-octet message says 'Hi there, server'
Reply sent, socket closed
Listening at ('127.0.0.1', 1060)
```

The foregoing example output is produced by having the client make one connection to the server, like this:

```
$ python tcp_sixteen.py client
Client has been assigned socket name ('127.0.0.1', 58185)
The server said 'Farewell, client'
```

You can see from the rest of the server code that, once a connected socket has been returned by `accept()`, it works exactly like a client socket with no further asymmetries evident in their pattern of communication. The `recv()` call returns data as it becomes available, and `sendall()` is the best way to send a new string of data when you want to make sure that it all gets transmitted.

You will note that an integer argument was provided to `listen()` when it was called on the server socket. This number indicates how many waiting connections, which have not yet had sockets created for them by `accept()` calls, should be allowed to stack up before the operating system starts turning new connections away by returning connection errors. We are using the very small value `1` here in our examples because we support only one example client connecting at a time; but we will consider larger values for this call when we talk about network server design in Chapter 7.

Once the client and server have said everything that they need to, they `close()` their end of the socket, which tells the operating system to transmit any remaining data still left in their output buffer and then conclude the TCP session with the shutdown procedure mentioned previously.

Address Already in Use

There is one last detail in Listing 3–1 that you might be curious about: why is the server careful to set the socket option `SO_REUSEADDR` before trying to bind to its port?

You can see the consequences of failing to set this option if you comment out that line and then try running the server. At first, you might think that it has no consequence. If all you are doing is stopping and starting the server, then you will see no effect at all:

```
$ python tcp_sixteen.py server
Listening at ('127.0.0.1', 1060)
^C
Traceback (most recent call last):
  ...
KeyboardInterrupt
$ python tcp_sixteen.py server
Listening at ('127.0.0.1', 1060)
```

But you will see a big difference if you bring up the server, run the client against it, and then try killing and re-running the server. When the server starts back up, you will get an error:

```
$ python tcp_sixteen.py server
Traceback (most recent call last):
  ...
socket.error: [Errno 98] Address already in use
```

How mysterious! Why would a `bind()` that can be repeated over and over again at one moment suddenly become impossible the next? If you keep trying to run the server without the `SO_REUSEADDR` option, you will find that the address does not become available again until several minutes after your last client connection!

The answer is that, from the point of view of your operating system's network stack, a socket that is merely *listening* can immediately be shut down and forgotten about, but a *connected* TCP socket—that is actually talking to a client—cannot immediately disappear when both ends have closed their connection and initiated the FIN handshakes in each direction. Why? Because after it sends the very last ACK packet,

the system has no way to ever be sure that it was received. If it was dropped by the network somewhere along its route, then the remote end might at any moment wonder what is taking the last ACK packet so long and re-transmit its FIN packet in the hope of finally receiving an answer.

A reliable protocol like TCP obviously has to have some point like this where it stops talking; some final packet must, logically, be left hanging with no acknowledgment, or systems would have to commit to an endless exchange of “okay, we both agree that we are all done, right?” messages until the machines were finally powered off. Yet even the final packet might get lost and need to be re-transmitted a few times before the other end finally receives it. What is the solution?

The answer is that once a connected TCP connection is finally closed from the point of view of your application, the operating system’s network stack actually keeps it around for up to four minutes in a waiting state (the RFC names these states CLOSE-WAIT and TIME-WAIT) so that any final FIN packets can be properly replied to. If instead the TCP implementation just forgot about the connection, then it could not reply to the FIN with a proper ACK.

So a server that tries claiming a port on which a live connection was running within the last few minutes is, really, trying to claim a port that is in some sense still in use. That is why you are returned an error if you try a `bind()` to that address. By specifying the socket option `SO_REUSEADDR`, you are indicating that your application is okay about owning a port whose old connections might still be shutting down out on some client on the network. In practice, I always use `SO_REUSEADDR` when writing server code without putting thought into it, and have never had any problems.

Binding to Interfaces

As was explained in Chapter 2 when we discussed UDP, the IP address that you pair with a port number when you perform a `bind()` operation tells the operating system which network interfaces you are willing to receive connections from. The example invocations of Listing 3–1 used the localhost IP address `127.0.0.1`, which protects your code from connections originating on other machines.

You can verify this by running Listing 3–1 in server mode as shown previously, and trying to connect with a client from another machine:

```
$ python tcp_sixteen.py client 192.168.5.130
Traceback (most recent call last):
  ...
socket.error: [Errno 111] Connection refused
```

You will see that the server Python code does not even react; the operating system does not even inform it that an incoming connection to its port was refused. (Note that if you have a firewall running on your machine, the client might just hang when it tries connecting, rather than getting a friendly “Connection refused” that tells it what is going on!)

But if you run the server with an empty string for the hostname, which tells the Python `bind()` routine that you are willing to accept connections through any of your machine’s active network interfaces, then the client can connect successfully from another host (the empty string is supplied by giving the shell these two double-quotes at the end of the command line):

```
$ python tcp_sixteen.py server ""
Listening at ('0.0.0.0', 1060)
We have accepted a connection from ('192.168.5.10', 46090)
Socket connects ('192.168.5.130', 1060) and ('192.168.5.10', 46090)
The incoming sixteen-octet message says 'Hi there, server'
Reply sent, socket closed
Listening at ('0.0.0.0', 1060)
```

As before, my operating system uses the special IP address `0.0.0.0` to mean "accept connections on any interface," but that may vary with operating system, and Python hides this fact by letting you use the empty string instead.

Deadlock

The term "deadlock" is used for all sorts of situations in computer science where two programs, sharing limited resources, can wind up waiting on each other forever because of poor planning. It turns out that it can happen fairly easily when using TCP.

I mentioned previously that typical TCP stacks use buffers, both so that they have somewhere to place incoming packet data until an application is ready to read it, and so that they can collect outgoing data until the network hardware is ready to transmit an outgoing packet. These buffers are typically quite limited in size, and the system is not generally willing to let programs fill all of RAM with unsent network data. After all, if the remote end is not yet ready to process the data, it makes little sense to expend system resources on the generating end trying to produce more of it.

This limitation will generally not trouble you if you follow the client-server pattern shown in Listing 3–1, where each end always reads its partner's complete message before turning around and sending data in the other direction. But you can run into trouble very quickly if you design a client and server that leave too much data waiting without having some arrangement for promptly reading it.

Take a look at Listing 3–2 for an example of a server and client that try to be a bit too clever without thinking through the consequences. Here, the server author has done something that is actually quite intelligent. His job is to turn an arbitrary amount of text into uppercase. Recognizing that its client's requests can be arbitrarily large, and that one could run out of memory trying to read an entire stream of input before trying to process it, the server reads and processes small blocks of 1,024 bytes at a time.

Listing 3–2. *TCP Server and Client That Deadlock*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 3 - tcp_deadlock.py
# TCP client and server that leave too much data waiting

import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = '127.0.0.1'
PORT = 1060

if sys.argv[1:] == ['server']:
»   s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
»   s.bind((HOST, PORT))
»   s.listen(1)
»   while True:
»   »   print 'Listening at', s.getsockname()
»   »   sc, sockname = s.accept()
»   »   print 'Processing up to 1024 bytes at a time from', sockname
»   »   n = 0
»   »   while True:
»   »   »   message = sc.recv(1024)
»   »   »   if not message:
»   »   »   »   break
»   »   »   sc.sendall(message.upper())  # send it back uppercase
»   »   »   n += len(message)
```

```
            print '\r%d bytes processed so far' % (n,),
            sys.stdout.flush()
        print
        sc.close()
        print 'Completed processing'

elif len(sys.argv) == 3 and sys.argv[1] == 'client' and sys.argv[2].isdigit():

    bytes = (int(sys.argv[2]) + 15) // 16 * 16  # round up to // 16
    message = 'capitalize this!'  # 16-byte message to repeat over and over

    print 'Sending', bytes, 'bytes of data, in chunks of 16 bytes'
    s.connect((HOST, PORT))

    sent = 0
    while sent < bytes:
        s.sendall(message)
        sent += len(message)
        print '\r%d bytes sent' % (sent,),
        sys.stdout.flush()

    print
    s.shutdown(socket.SHUT_WR)

    print 'Receiving all the data the server sends back'

    received = 0
    while True:
        data = s.recv(42)
        if not received:
            print 'The first data received says', repr(data)
        received += len(data)
        if not data:
            break
        print '\r%d bytes received' % (received,),

    s.close()

else:
    print >>sys.stderr, 'usage: tcp_deadlock.py server | client <bytes>'
```

It can split the work up so easily, by the way, because it is merely trying to run the `upper()` string method on plain ASCII characters—an operation that can be performed separately on each block of input, without worrying about the blocks that came before or after. Things would not be this simple for the server if it were trying to run a more sophisticated string operation like `title()`, which would capitalize a letter in the middle of a word if the word happened to be split across a block boundary. For example, if a message got split into16-byte blocks, then errors would creep in like this:

```
>>> message = 'the tragedy of macbeth'
>>> blocks = message[:16], message[16:]
>>> ''.join( b.upper() for b in blocks )   # works fine
'THE TRAGEDY OF MACBETH'
>>> ''.join( b.title() for b in blocks )   # whoops
'The Tragedy Of MAcbeth'
```

Processing text while splitting on fixed-length blocks will also not work for UTF-8 encoded Unicode data, since a multi-byte character could get split across a boundary between two of the binary blocks. In both cases, the server would have to be more careful, and carry some state between one block of data and the next.

In any case, handling input a block at a time like this is quite smart for the server, even if the 1,024-byte block size, used here for illustration, is actually a very small value for today's servers and networks. By handling the data in pieces and immediately sending out responses, the server limits the amount of data that it actually has to keep in memory at any one time. Servers designed like this could handle hundreds of clients at once, each sending streams totaling gigabytes, without taxing the memory resources of the server machine.

And, for small data streams, the client and server in Listing 3–2 seem to work fine. If you start the server and then run the client with a command-line argument specifying a modest number of bytes—say, asking it to send 32 bytes of data (for simplicity, it will round whatever value you supply up to a multiple of 16 bytes)—then it will get its text back in all uppercase:

```
$ python tcp_deadlock.py client 32
Sending 32 bytes of data, in chunks of 16 bytes
32 bytes sent
Receiving all the data the server sends back
The first data received says 'CAPITALIZE THIS!CAPITALIZE THIS!'
32 bytes received
```

The server (which, by the way, needs to be running on the same machine—this script uses the localhost IP address to make the example as simple as possible) will report that it indeed processed 32 bytes on behalf of its recent client:

```
$ python tcp_deadlock.py server
Processing up to 1024 bytes at a time from ('127.0.0.1', 46400)
32 bytes processed so far
Completed processing
Listening at ('127.0.0.1', 1060)
```

So this code works well for small amounts of data. In fact, it might also work for larger amounts; try running the client with hundreds or thousands of bytes, and see whether it continues to work.

This first example exchange of data, by the way, shows you the behavior of `recv()` that I have previously described: even though the server asked for 1,024 bytes to be received, `recv(1024)` was quite happy to return only 16 bytes, if that was the amount of data that became available and no further data had yet arrived from the client.

But if you try a large enough value, then disaster strikes! Try using the client to send a very large stream of data, say, one totaling a gigabyte:

```
$ python tcp_deadlock.py client 1073741824
```

You will see both the client and the server furiously updating their terminal windows as they breathlessly update you with the amount of data they have transmitted and received. The numbers will climb and climb until, quite suddenly, both connections freeze. Actually, if you watch carefully, you will see the server stop first, and then the client will grind to a halt soon afterward. The amount of data processed before they seize up varies on the Ubuntu laptop on which I am typing this chapter, but on the test run that I just completed here on my laptop, the Python script stopped with the server saying:

```
$ python tcp_deadlock.py server
Listening at ('127.0.0.1', 1060)
Processing up to 1024 bytes at a time from ('127.0.0.1', 46411)
602896 bytes processed so far
```

And the client is frozen about 100,000 bytes farther ahead in writing its outgoing data stream:

```
$ python tcp_deadlock.py client 1073741824
Sending 1073741824 bytes of data, in chunks of 16 bytes
734816 bytes sent
```

Why have both client and server been brought to a halt?

The answer is that the server's output buffer and the client's input buffer have both finally filled, and TCP has used its window adjustment protocol to signal this fact and stop the socket from sending more data that would have to be discarded and later re-sent.

Consider what happens as each block of data travels. The client sends it with `sendall()`. Then the server accepts it with `recv()`, processes it, and then transmits its capitalized version back out with another `sendall()` call. And then what? Well, nothing! The client is never running any `recv()` calls—not while it still has data to send—so more and more capitalized data backs up, until the operating system is not willing to accept any more.

During the run shown previously, about 600KB was buffered by the operating system in the client's incoming queue before the network stack decided that it was full. At that point, the server blocks in its `sendall()` call, and is paused there by the operating system until the logjam clears and it can send more data. With the server no longer processing data or running any more `recv()` calls, it is now the client's turn to have data start backing up. The operating system seems to have placed a limit of around 130KB to the amount of data it would queue up in that direction, because the client got roughly another 130KB into producing the stream before finally being brought to a halt as well.

On a different system, you will probably find that different limits are reached. So the foregoing numbers are arbitrary and based on the mood of my laptop at the moment; they are not at all inherent in the way TCP works.

And the point of this example is to teach you two things—besides, of course, showing that `recv(1024)` indeed returns fewer bytes than 1,024 if a smaller number are immediately available!

First, this example should make much more concrete the idea that there are buffers sitting inside the TCP stacks on each end of a network connection. These buffers can hold data temporarily so that packets do not have to be dropped and eventually re-sent if they arrive at a moment that their reader does not happen to be inside of a `recv()` call. But the buffers are not limitless; eventually, a TCP routine trying to write data that is never being received or processed is going to find itself no longer able to write, until some of the data is finally read and the buffer starts to empty.

Second, this example makes clear the dangers involved in protocols that do not alternate lock-step between the client requesting and the server acknowledging. If a protocol is not strict about the server reading a complete request until the client is done sending, and then sending a complete response in the other direction, then a situation like that created here can cause both of them to freeze without any recourse other than killing the program manually, and then rewriting it to improve its design!

But how, then, are network clients and servers supposed to process large amounts of data without entering deadlock? There are, in fact, two possible answers: either they can use socket options to turn off blocking, so that calls like `send()` and `recv()` return immediately if they find that they cannot send any data yet. We will learn more about this option in Chapter 7, where we look in earnest at the possible ways to architect network server programs.

Or, the programs can use one of several techniques to process data from several inputs at a time, either by splitting into separate threads or processes—one tasked with sending data into a socket, perhaps, and another tasked with reading data back out—or by running operating system calls like `select()` or `poll()` that let them wait on busy outgoing and incoming sockets at the same time, and respond to whichever is ready.

Finally, note carefully that the foregoing scenario cannot ever happen when you are using UDP! This is because UDP does not implement flow control. If more datagrams are arriving up than can be processed, then UDP can simply discard some of them, and leave it up to the application to discover that they went missing.

Closed Connections, Half-Open Connections

There are two more points that should be made, on a different subject, from the foregoing example.

First, Listing 3–2 shows us how a Python socket object behaves when an end-of-file is reached. Just like a Python file object returns an empty string upon a `read()` when there is no more data left, a socket simply returns an empty string when the socket is closed.

We never worried about this in Listing 3–1 because in that case we had imposed a strict enough structure on our protocol—exchanging a pair of messages of exactly 16 bytes—that we did not need to close the socket to signal when communication was done. The client and server simply sent their messages, and then could close their sockets separately without needing to do any further checks.

But in Listing 3–2, the client sends—and thus the server also processes and sends back—an arbitrary amount of data whose length is not decided until the user enters a number of bytes on the command line. And so you can see in the code, twice, the same pattern: a `while` loop that runs until it finally sees an empty string returned from `recv()`. Note that this normal Pythonic pattern will *not* work once we reach Chapter 7 and explore non-blocking sockets—in that case, `recv()` might return an empty string simply because no data is available at the moment, and other techniques are used to determine whether the socket has closed.

Second, you will see that the client makes a `shutdown()` call on the socket after it finishes sending its transmission. This solves an important problem: if the server is going to read forever until it sees end-of-file, then how will the client avoid having to do a full `close()` on the socket and thus forbid itself from doing the many `recv()` calls that it still needs to make to receive the server's response? The solution is to "half-close" the socket—that is, to permanently shut down communication in one direction but without destroying the socket itself—so that the server can no longer read any data, but can still send any remaining reply back in the other direction, which will still be open.

The `shutdown()` call can be used to end either direction of communication in a two-way socket like this; its argument can be one of three symbols:

- `SHUT_WR`: This is the most common value used, since in most cases a program knows when its own output is finished but not about when its conversation partner will be done. This value says that the caller will be writing no more data into the socket, and that reads from its other end should act like it is closed.
- `SHUT_RD`: This is used to turn off the incoming socket stream, so that an end-of-file error is encountered if your peer tries to send any more data to you on the socket.
- `SHUT_RDWR`: This closes communication in both directions on the socket. It might not, at first, seem useful, because you can also just perform a `close()` on the socket and communication is similarly ended in both directions. The difference is a rather advanced one: if several programs on your operating system are allowed to share a single socket, then `close()` just ends your process's relationship with the socket, but keeps it open as long as another process is still using it; but `shutdown()` will always immediately disable the socket for everyone using it.

Since you are not allowed to create unidirectional sockets through a standard `socket()` call, many programmers who need to send information only in one direction over a socket will first create the socket, then—as soon as it is connected—immediately run `shutdown()` for the direction that they do not need. This means that no operating system buffers will be needlessly filled if the peer with which they are communicating accidentally tries to send data in a direction that it should not.

Running `shutdown()` immediately on sockets that should really be unidirectional also provides a more obvious error message for a peer that does get confused and tries to send data. Otherwise their data will either simply be ignored, or might even fill a buffer and cause a deadlock because it will never be read.

Using TCP Streams like Files

Since TCP supports streams of data, they might have already reminded you of normal files, which also support reading and writing as fundamental operations. Python does a very good job of keeping these concepts separate: file objects can `read()` and `write()`, sockets can `send()` and `recv()`, and no kind of object can do both. This is actually a substantially cleaner conceptual split than is achieved by the underlying POSIX interface, which lets a C programmer call `read()` and `write()` on a socket indiscriminately as though it were a normal file descriptor!

But sometimes you will want to treat a socket like a normal Python file object—often because you want to pass it into code like that of the many Python modules such as `pickle`, `json`, and `zlib` that can read and write data directly from a file. For this purpose, Python provides a `makefile()` method on every socket that returns a Python file object that is really calling `recv()` and `send()` behind the scenes:

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> hasattr(s, 'read')
False
>>> f = s.makefile()
>>> hasattr(f, 'read')
True
```

Sockets, like normal Python files, also have a `fileno()` method that lets you discover their file descriptor number in case you need to supply it to lower-level calls; we will find this very helpful when we explore `select()` and `poll()` in Chapter 7.

Summary

The TCP-powered "stream" socket does whatever is necessary—including re-transmitting lost packets, reordering the ones that arrive out of sequence, and splitting very large data streams into optimally sized packets for your network in the first place—to support the transmission and reception of streams of data over the network between two sockets.

As with UDP, port numbers are used by TCP to distinguish the many stream endpoints that might exist on a single machine. A program that wants to accept incoming TCP connections needs to `bind()` to a port, run `listen()` on the socket, and then go into a loop that runs `accept()` over and over to receive a new socket for each incoming connection with which it can talk to each particular client that connects. Programs that want to connect to existing server ports need only create a socket and `connect()` to an address.

Servers will usually want to set the `SO_REUSEADDR` option on the sockets they `bind()`, lest old connections still closing down on the same port from the last time the server was run prevent the operating system from allowing the binding.

Data is actually sent and received with `send()` and `recv()`. Some protocols will mark up their data so that clients and servers know automatically when a communication is complete. Other protocols will treat the TCP socket as a true stream and send and receive until end-of-file is reached. The `shutdown()` socket method can be used to produce end-of-file in one direction on a socket (all sockets are bidirectional by nature) while leaving the other direction open.

Deadlock can occur if two peers are written such that the socket fills with more and more data that never gets read. Eventually, one direction will no longer be able to `send()` and might hang forever waiting for the backlog to clear.

If you want to pass a socket to a Python routine that knows how to read to or write from a normal file object, the `makefile()` socket method will give you a Python object that calls `recv()` and `send()` behind the scenes when the caller needs to read and write.

CHAPTER 4

Socket Names and DNS

Having spent the last two chapters learning the basics of UDP and TCP, the two major data transports available on IP networks, it is time for us to step back and talk about two larger issues that need to be tackled regardless of which transport you are using. In this chapter, we will discuss the topic of network addresses and will describe the distributed service that allows names to be resolved to raw IP addresses.

Hostnames and Domain Names

Before we plunge into this topic, we should get a few terms straight that will play a big role in the discussion that follows.

- *Top-level domain (TLD):* These are the few hundred strings like `com`, `net`, `org`, `gov`, and `mil` that, together with country codes like `de` and `uk`, form the set of possible suffixes for valid domain names. Typically, each TLD has its own set of servers and its own organization that is in charge of granting ownership to domains beneath the TLD.
- *Domain name:* This is the name that a business or organization appends as a suffix to its sites and hosts on the Internet, like `python.org`, `imdb.com`, or `bbc.co.uk`. It typically costs an annual fee to own a domain name, but owning one gives you the right to create as many hostnames beneath it as you want.
- *Fully qualified domain name:* The FQDN names an Internet site or host by appending its organization's full domain name to the name of a particular machine in that organization. Example FQDNs are `gnu.org` and `asaph.rhodesmill.org`. Whether a domain name is "fully qualified" does not depend on its having any specific number of components—it may have two, three, four, or more dot-separated names. What makes it a FQDN is that it ends with a TLD and therefore will work from anywhere. You can often use just the hostname `athena` if you are connected to an MIT network, but from anywhere else in the world, you have to fully qualify the name and specify `athena.mit.edu`.
- *Hostname:* This term, unfortunately, is ambiguous! Sometimes it means the bare, unqualified name that a machine might print when you connect to it, like `asaph` or `athena`. But sometimes people instead mean the FQDN when they say "the hostname."

- In general, an FQDN may be used to identify a host from anywhere else on the Internet. Bare hostnames, by contrast, work as relative names only if you are already inside the organization and using their own nameservers (a concept we discuss later in this chapter) to resolve names on your desktop, laptop, or server. Thus `athena` should work as an abbreviation for `athena.mit.edu` if you are actually on the MIT campus, but it will not work if you are anywhere else in the world—unless you have configured your system to always try MIT hostnames first, which would be unusual, but maybe you are on their staff or something.

Socket Names

The last two chapters have already introduced you to the fact that sockets cannot be named with a single primitive Python value like a number or string. Instead, both TCP and UDP use integer port numbers to share a single machine's IP address among the many different applications that might be running there, and so the address and port number have to be combined in order to produce a socket name, like this:

```
('18.9.22.69', 80)
```

While you might have been able to pick up some scattered facts about socket names from the last few chapters—like the fact that the first item can be either a hostname or a dotted IP address—it is time for us to approach the whole subject in more depth.

You will recall that socket names are important at several points in the creation and use of sockets. For your reference, here are all of the major socket methods that demand of you some sort of socket name as an argument:

- `mysocket.accept()`: Each time this is called on a listening TCP stream socket that has incoming connections ready to hand off to the application, it returns a tuple whose second item is the remote address that has connected (the first item in the tuple is the net socket connected to that remote address).
- `mysocket.bind(address)`: Assigns the socket the local `address` so that outgoing packets have an address from which to originate, and so that any incoming connections from other machines have a name that they can use to connect.
- `mysocket.connect(address)`: Establishes that data sent through this socket will be directed to the given remote `address`. For UDP sockets, this simply sets the default address used if the caller uses `send()` rather than `sendto()`; for TCP sockets, this actually negotiates a new stream with another machine using a three-way handshake, and raises an exception if the negotiation fails.
- `mysocket.getpeername()`: Returns the remote address to which this socket is connected.
- `mysocket.getsockname()`: Returns the address of this socket's own local endpoint.
- `mysocket.recvfrom(...)`: For UDP sockets, this returns a tuple that pairs a string of returned data with the address from which it was just sent.
- `mysocket.sendto(data, address)`: An unconnected UDP port uses this method to fire off a data packet at a particular remote address.

So, there you have it! Those are the major socket operations that care about socket addresses, all in one place, so that you have some context for the remarks that follow. In general, any of the foregoing methods can receive or return any of the sorts of addresses that follow, meaning that they will work

regardless of whether you are using IPv4, IPv6, or even one of the less common address families that we will not be covering in this book.

Five Socket Coordinates

Monty Python's *Holy Grail* famously includes "the aptly named Sir Not-Appearing-In-This-Film" in its list of knights of the round table, and this section does something of the same service for this book. Here we will consider the full range of "coordinates" that identify a socket, only to note that most of the possible values are not within the scope of our project here in this book.

When reviewing the sample programs of Chapter 2 and Chapter 3, we paid particular attention to the hostnames and IP addresses that their sockets used. But if you read each program listing from the beginning, you will see that these are only the last two coordinates of five major decisions that were made during the construction and deployment of each socket object. Recall that the steps go something like this:

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> s.bind(('localhost', 1060))
```

In order, here is the full list of values that had to be chosen, and you will see that there are five in all.

First, the *address family* makes the biggest decision: it names what kind of network you want to talk to, out of the many kinds that a particular machine might support.

In this book, we will always use the value `AF_INET` for the address family, because we believe that making this book about IP networking will best serve the vast majority of Python programmers, while at the same time giving them skills that will work on Linux, Mac OS, or even Windows. But if you will import the `socket` module in Python, print out `dir(socket)`, and look for the symbols that start with `AF_` ("Address Family"), you may see choices whose names you will recognize, like AppleTalk and Bluetooth.

Especially popular on POSIX systems is the `AF_UNIX` address family, which works just like Internet sockets but runs directly between programs on the same machine with more efficiency than is possible when traversing the entire IP network stack just to arrive back at the localhost interface.

Next after the address family comes the *socket type*. It chooses the particular kind of communication technique that you want to use on the network you have chosen. You might guess that every single address family presents entirely different socket types that you would have to go look up for each one, since, after all, what address family besides `AF_INET` is going to present socket types like UDP and TCP?

Happily, this suspicion is misplaced. Although UDP and TCP are indeed quite specific to the `AF_INET` protocol family, the socket interface designers decided to create more generic names for the broad idea of a packet-based socket, which goes by the name `SOCK_DGRAM`, and the broad idea of a reliable flow-controlled data stream, which as we have seen is known as a `SOCK_STREAM`. Because many address families support either one or both of these mechanisms, even though they might implement them a bit differently than they are implemented under IP, only these two symbols are necessary to cover many protocols under a variety of different address families.

The third field in the `socket()` call, the *protocol*, is rarely used because once you have specified the address family and socket type, you have narrowed down the possible protocols to one major option. For this reason, programmers usually leave this unspecified or provide the value zero to force it to be chosen automatically. If you want a stream under IP, the system knows to choose TCP; if you want datagrams, then it selects UDP. That is why none of our `socket()` calls in this book has a third argument: it is in practice almost never needed. Look inside the `socket` module for names starting with `IPPROTO` for some examples of protocols defined for the `AF_INET` family; listed there you will see the two this book actually addresses, under the names `IPPROTO_TCP` and `IPPROTO_UDP`.

The fourth and fifth fields are, then, the IP address and UDP or TCP port number that were explained in detail in the last two chapters.

But we should immediately step back, and note that it is only because of our specific choices for the first three coordinates that our socket names have had two components, hostname and port! If you instead had chosen AppleTalk or ATM or Bluetooth for your address family, then some other data structure might have been required of you instead of a tuple with a string and an integer inside. So the whole set of coordinates, which in this section we have talked about as five coordinates, is really three fixed coordinates needed to create the socket, followed by however many more coordinates your particular address family requires you to use in order to make a network connection.

IPv6

And having explained all of that, it turns out that this book actually does need to introduce one additional address family, beyond the `AF_INET` we have used so far: the address family for IPv6, named `AF_INET6`, which is the way forward into a future where the world does not, in fact, run out of IP addresses.

Once the old ARPANET really started taking off, its choice of 32-bit address names—which made so much sense back when computer memory chips were measured by the kilobyte—became a clear and worrying limitation. With only about four billion possible addresses available, even assuming that we could use the address space that fully, that makes fewer than one IP address for every person on the earth—which means real trouble once everyone has *both* a computer and an iPhone!

Even though only a few percent of the computers on the Internet today are actually using IPv6 to communicate with the global network through their Internet service providers (where "today" is September 2010), the steps necessary to make your Python programs compatible with IPv6 are simple enough that you should go ahead and try writing code that prepares us all for the future.

In Python you can test directly for whether the underlying platform supports IPv6 by checking the `has_ipv6` Boolean attribute inside the `socket` module:

```
>>> import socket
>>> socket.has_ipv6
True
```

But note that this does *not* tell you whether an actual IPv6 interface is up and configured and can currently be used to send packets anywhere; it is purely an assertion about whether IPv6 support has been compiled into the operating system, not about whether it is in use!

The differences that IPv6 will make for your Python code might sound quite daunting, if listed one right after the other:

- Your sockets have to be prepared to have the family `AF_INET6` if you are called upon to operate on an IPv6 network.
- No longer do socket names consist of just two pieces, an address and a port number; instead, they can also involve additional coordinates that provide "flow" information and a "scope" identifier.
- The pretty IPv4 octets like `18.9.22.69` that you might already be reading from configuration files or from your command-line arguments will now sometimes be replaced by IPv6 host addresses instead, which you might not even have good regular expressions for yet. They have lots of colons, they can involve hexadecimal numbers, and in general they look quite ugly.

The benefits of the IPv6 transition are not only that it will make an astronomically large number of addresses available, but also that the protocol has more complete support for things like link-level security than do most implementations of IPv4.

But the changes just listed can sound like a lot of trouble if you have been in the habit of writing clunky, old-fashioned code that puts IP addresses and hostnames through regular expressions of your

own devising. If, in other words, you have been in the business of interpreting addresses yourself in any form, you probably imagine that the transition to IPv6 will make you write even more complicated code than you already have. Fear not: my actual recommendation is that you get out of address interpretation or scanning altogether, and the next section will show you how!

Modern Address Resolution

To make your code simple, powerful, and immune from the complexities of the transition from IPv4 to IPv6, you should turn your attention to one of the most powerful tools in the Python socket user's arsenal: `getaddrinfo()`.

The `getaddrinfo()` function sits in the `socket` module along with most other operations that involve addresses (rather than being a socket method). Unless you are doing something specialized, it is probably the only routine that you will ever need to transform the hostnames and port numbers that your users specify into addresses that can be used by `socket` methods!

Its approach is simple: rather than making you attack the addressing problem piecemeal, which is necessary when using the older routines in the `socket` module, it lets you specify everything you know about the connection that you need to make in a single call. In response, it returns all of the coordinates we discussed earlier that are necessary for you to create and connect a socket to the named destination.

Its basic use is very simple and goes like this:

```
>>> from pprint import pprint
>>> infolist = socket.getaddrinfo('gatech.edu', 'www')
>>> pprint(infolist)
[(2, 1, 6, '', ('130.207.244.244', 80)),
 (2, 2, 17, '', ('130.207.244.244', 80))]
>>> ftpca = infolist[0]
>>> ftpca[0:3]
(2, 1, 6)
>>> s = socket.socket(*ftpca[0:3])
>>> ftpca[4]
('130.207.244.244', 80)
>>> s.connect(ftpca[4])
```

The variable that I have so obscurely named `ftpca` here is an acronym for the order of the variables that are returned: "family, type, protocol, canonical name, and address," which contain everything you need to make a connection. Here, we have asked about the possible methods for connecting to the HTTP port of the host `gatech.edu`, and have been told that there are two ways to do it: by creating a `SOCK_STREAM` socket (socket type 1) that uses `IPPROTO_TCP` (protocol number 6) or else by using a `SOCK_DGRAM` (socket type 2) socket with `IPPROTO_UDP` (which is the protocol represented by the integer17).

And, yes, the foregoing answer is indicative of the fact that HTTP officially supports both TCP and UDP, at least according to the official organization that doles out port numbers (and that gave HTTP one of each). Usually when calling `getaddrinfo()`, you will specify which kind of socket you want rather than leaving the answer up to chance!

If you use `getaddrinfo()` in your code, then unlike the listings in Chapter 2 and Chapter 3—which used real symbols like `AF_INET` just to make it clearer how the low-level socket mechanisms were working—your production Python code might not even have to reference any symbols at all from the `socket` module except for those that explain to `getaddrinfo()` which kind of address you want. Instead, you will use the first three items in the `getaddrinfo()` return value as the arguments to the `socket()` constructor, and then use the fifth item as the address to any of the calls listed in the first section of this chapter.

As you can see from the foregoing code snippet, `getaddrinfo()` generally allows not only the hostname but also the port name to be a symbol rather than an integer—eliminating the need of older

Python code to make extra calls if the user might want to provide a symbolic port number like `www` or `smtp` instead of 80 or 25.

Asking getaddrinfo() Where to Bind

Before tackling all of the options that `getaddrinfo()` supports, it will be more useful to see how it is used to support three basic network operations. We will tackle them in the order that you might perform operations on a socket: binding, connecting, and then identifying a remote host who has sent you information.

If you want an address to provide to `bind()`, either because you are creating a server socket or because you for some reason want your client to be connecting to someone else but from a predictable address, then you will call `getaddrinfo()` with `None` as the hostname but with the port number and socket type filled in. Note that here, as in the following `getaddrinfo()` calls, zeros serve as wildcards in fields that are supposed to contain numbers:

```
>>> from socket import getaddrinfo
>>> getaddrinfo(None, 'smtp', 0, socket.SOCK_STREAM, 0, socket.AI_PASSIVE)
[(2, 1, 6, '', ('0.0.0.0', 25)), (10, 1, 6, '', ('::', 25, 0, 0))]
>>> getaddrinfo(None, 53, 0, socket.SOCK_DGRAM, 0, socket.AI_PASSIVE)
[(2, 2, 17, '', ('0.0.0.0', 53)), (10, 2, 17, '', ('::', 53, 0, 0))]
```

Here we asked about where we should `bind()` a socket if we want to serve SMTP traffic using TCP, and if we want to serve DNS traffic using DCP, respectively. The answers we got back in each case are the appropriate wildcard addresses that will let us bind to every IPv4 and every IPv6 interface on the local machine with all of the right values for the socket family, socket type, and protocol in each case.

If you instead want to `bind()` to a particular IP address that you know that the local machine holds, then omit the `AI_PASSIVE` flag and just specify the hostname. For example, here are two ways that you might try binding to localhost:

```
>>> getaddrinfo('127.0.0.1', 'smtp', 0, socket.SOCK_STREAM, 0)
[(2, 1, 6, '', ('127.0.0.1', 25))]
>>> getaddrinfo('localhost', 'smtp', 0, socket.SOCK_STREAM, 0)
[(10, 1, 6, '', ('::1', 25, 0, 0)), (2, 1, 6, '', ('127.0.0.1', 25))]
```

You can see that supplying the IPv4 address for the localhost locks you down to receiving connections only over IPv4, while using the symbolic name `localhost` (at least on my Linux laptop, with a well-configured `/etc/hosts` file) makes available both the IPv4 and IPv6 local names for the machine.

One question that you might already be asking at this point, by the way, is what on earth you are supposed to do when you assert that you want to supply a basic service and `getaddrinfo()` goes and gives you several addresses to use—you certainly cannot create a single socket and `bind()` it to more than one address! In Chapter 7, we will tackle the techniques that you can use if you are writing server code and want to have several sockets going at once.

Asking getaddrinfo() About Services

Except for the use shown in the previous section, all other uses of `getaddrinfo()` are outward-looking, and generate information suitable for connecting you to other applications. In all such cases, you can either use an empty string to indicate that you want to connect back to the localhost using the loopback interface, or provide a string giving an IPv4 address, IPv6 address, or hostname to name your destination.

The usual use of `getaddrinfo()` in all other cases—which, basically, is when you are preparing to `connect()` or `sendto()`—is to specify the `AI_ADDRCONFIG` flag, which filters out any addresses that are

impossible for your computer to reach. For example, an organization might have both an IPv4 and an IPv6 range of IP addresses; but if your particular host supports only IPv4, then you will want the results filtered to include only addresses in that family. In case the local machine has only an IPv6 network interface but the service you are connecting to is supporting only IPv4, the AI_V4MAPPED will return you those IPv4 addresses re-encoded as IPv6 addresses that you can actually use.

So you will usually use getaddrinfo() this way when connecting:

```
>>> getaddrinfo('ftp.kernel.org', 'ftp', 0, socket.SOCK_STREAM, 0,
...     socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
[(2, 1, 6, '', ('204.152.191.37', 21)),
 (2, 1, 6, '', ('149.20.20.133', 21))]
```

And we have gotten exactly what we wanted: every way to connect to a host named ftp.kernel.org through a TCP connection to its FTP port. Note that several IP addresses were returned because this service, to spread load, is located at several different machines on the Internet. You should generally always use the first address returned, and if a connection fails, then try the remaining ones, because there is intelligence built into the name-resolution system to properly randomize the order in which you receive them. By always trying the first server IP address first, you will offer the various servers a workload that is in the proportion that the machine administrators intend.

Here is another query, which describes how I can connect from my laptop to the HTTP interface of the IANA that assigns port numbers in the first place:

```
>>> getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0,
...     socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
[(2, 1, 6, '', ('192.0.43.8', 80))]
```

The IANA web site is actually a good one for demonstrating the utility of the AI_ADDRCONFIG flag, because—like any other good Internet standards organization—their web site already supports IPv6. It just so happens that my laptop can speak only IPv4 on the wireless network to which it is currently connected, so the foregoing call was careful to return only an IPv4 address. But if we take away our carefully chosen flags in the sixth parameter, then we will also be able to see their IPv6 address:

```
>>> getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0)
[(2, 1, 6, '', ('192.0.43.8', 80)),
 (10, 1, 6, '', ('2001:500:88:200::8', 80, 0, 0))]
```

This can be useful if you are not going to try to use the addresses yourself, but if you are providing some sort of directory information to other hosts or programs.

Asking getaddrinfo() for Pretty Hostnames

One last circumstance that you will commonly encounter is where you either are making a new connection, or maybe have just received a connection to one of your own sockets, and you want an attractive hostname to display to the user or record in a log file. This is slightly dangerous because a hostname lookup can take quite a bit of time, even on the modern Internet, and might return a hostname that no longer works by the time you go and check your logs—so for log files, try to record both the hostname and raw IP address!

But if you have a good use for the "canonical name" of a host, then try running getaddrinfo() with the AI_CANONNAME flag turned on, and the fourth item of any of the tuples that it returns—that were always empty strings in the foregoing examples, you will note—will contain the canonical name:

```
>>> getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0,
...     socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME)
[(2, 1, 6, '43-8.any.icann.org', ('192.0.43.8', 80))]
```

You can also supply getaddrinfo() with the attributes of a socket that is already connected to a remote peer, and get a canonical name in return:

```
>>> mysock = old_sock.accept()
>>> addr, port = mysock.getpeername()
>>> getaddrinfo(addr, port, mysock.family, mysock.type, mysock.proto,
...             socket.AI_CANONNAME)
[(2, 1, 6, 'rr.pmtpa.wikimedia.org', ('208.80.152.2', 80))]
```

Again, this will work only if the owner of the IP address happens to have a name defined for it (and, obviously, it requires the hostname lookup to succeed).

Other getaddrinfo() Flags

The examples just given showed the operation of three of the most important getaddrinfo() flags. The flags available vary somewhat by operating system, and you should always consult your own computer's documentation (not to mention its configuration!) if you are confused about a value that it chooses to return. But there are several flags that tend to be cross-platform; here are some of the more important ones:

- AI_ALL: We have already discussed that the AI_V4MAPPED option will save you in the situation where you are on a purely IPv6-connected host, but the host to which you want to connect advertises only IPv4 addresses: it resolves this problem by "mapping" the IPv4 addresses to their IPv6 equivalent. But if some IPv6 addresses do happen to be available, then they will be the only ones shown. Thus the existence of this option: if you want to see all of the addresses from your IPv6-connected host, even though some perfectly good IPv6 addresses are available, then combine this AI_ALL flag with AI_V4MAPPED and the list returned to you will have every address known for the target host.

- AI_NUMERICHOST: This turns off any attempt to interpret the hostname parameter (the first parameter to getaddrinfo()) as a textual hostname like cern.ch, and only tries to interpret the hostname string as a literal IPv4 or IPv6 hostname like 74.207.234.78 or fe80::fcfd:4aff:fecf:ea4e. This is much faster, as no DNS round-trip is incurred (see the next section), and prevents possibly untrusted user input from forcing your system to issue a query to a nameserver under someone else's control.

- AI_NUMERICSERV: This turns off symbolic port names like www and insists that port numbers like 80 be used instead. This does not necessarily have the network-query implications of the previous option, since port-number databases are typically stored locally on IP-connected machines; on POSIX systems, resolving a symbolic port name typically requires only a quick scan of the /etc/services file (but check your /etc/nsswitch.conf file's services option to be sure). But if you know your port string should always be an integer, then activating this flag can be a useful sanity check.

One final note about flags: you do *not* have to worry about the IDN-related flags that some operating systems use in order to enable getaddrinfo() to resolve those fancy new domain names that have Unicode characters in them. Instead, Python will accept a Unicode string as the hostname and set whatever options are necessary to get it converted for you:

```
>>> getaddrinfo(u'παράδειγμα.δοκιμή', 'www', 0, socket.SOCK_STREAM, 0,
...     socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
[(2, 1, 6, '', ('199.7.85.13', 80))]
```

If you are curious about how this works behind the scenes, read up on the relevant international standards starting with RFC 3492, and note that Python now includes an `idna` codec that can translate to and from internationalized domain names:

```
>>> u'παράδειγμα.δοκιμή'.encode('idna')
'xn--hxajbheg2az3al.xn--jxalpdlp'
```

It is this resulting plain-ASCII string that is actually sent to the domain name service when you enter the Greek sample domain name just shown.

Primitive Name Service Routines

Before `getaddrinfo()` was all the rage, programmers doing socket-level programming got by with a simpler collection of name service routines supported by the operating system. They should be avoided today since most of them are hardwired to speak only IPv4.

You can find their documentation in the Standard Library page on the `socket` module. Here, the most efficient thing to do will be to play show-and-tell and use quick examples to illustrate each call. Two calls let you learn about the hostname of the current machine:

```
>>> socket.gethostname()
'asaph'
>>> socket.getfqdn()
'asaph.rhodesmill.org'
```

And two more let you convert between IPv4 hostnames and IP addresses:

```
>>> socket.gethostbyname('cern.ch')
'137.138.144.169'
>>> socket.gethostbyaddr('137.138.144.169')
('webr8.cern.ch', [], ['137.138.144.169'])
```

Finally, three routines let you look up protocol numbers and ports using symbolic names known to your operating system:

```
>>> socket.getprotobyname('UDP')
17
>>> socket.getservbyname('www')
80
>>> socket.getservbyport(80)
'www'
```

If you want to try learning the primary IP address for the machine on which your Python program is running, you can try passing its fully qualified hostname into a `gethostbyname()` call, like this:

```
>>> socket.gethostbyname(socket.getfqdn())
'74.207.234.78'
```

But since either call could fail and return an address error (see the section on error handling in Chapter 5), your code should have a backup plan in case this pair of calls fails to return a useful IP address.

Using getsockaddr() in Your Own Code

To put everything together, I have assembled a quick example of how `getaddrinfo()` looks in actual code. Take a look at Listing 4–1.

Listing 4–1. *Using `getaddrinfo()` to Create and Connect a Socket*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 4 - www_ping.py
# Find the WWW service of an arbitrary host using getaddrinfo().

import socket, sys

if len(sys.argv) != 2:
»   print >>sys.stderr, 'usage: www_ping.py <hostname_or_ip>'
»   sys.exit(2)

hostname_or_ip = sys.argv[1]

try:
»   infolist = socket.getaddrinfo(
»   »   hostname_or_ip, 'www', 0, socket.SOCK_STREAM, 0,
»   »   socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME,
»   »   )
except socket.gaierror, e:
»   print 'Name service failure:', e.args[1]
»   sys.exit(1)

info = infolist[0]  # per standard recommendation, try the first one
socket_args = info[0:3]
address = info[4]
s = socket.socket(*socket_args)
try:
»   s.connect(address)
except socket.error, e:
»   print 'Network failure:', e.args[1]
else:
»   print 'Success: host', info[3], 'is listening on port 80'
```

It performs a simple are-you-there test of whatever web server you name on the command line by attempting a quick connection to port 80 with a streaming socket. Using the script would look something like this:

```
$ python www_ping.py mit.edu
Success: host WEB.MIT.EDU is listening on port 80
$ python www_ping.py smtp.google.com
Network failure: Connection timed out
$ python www_ping.py no-such-host.com
Name service failure: No address associated with hostname
```

Note three things about the source code.

First, it is completely general, and contains no mention either of IP as a protocol nor of TCP as a transport. If the user happened to type a hostname that the system recognized as a host to which it was connected through AppleTalk (if you can imagine that sort of thing in this day and age), then

getaddrinfo() would be free to return the AppleTalk socket family, type, and protocol, and that would be the kind of socket that we would wind up creating and connecting.

Second, note that getaddrinfo() failures cause a specific name service error, which Python calls a gaierror, rather than a plain socket error of the kind used for the normal network failure that we detected at the end of the script. We will learn more about error handling in Chapter 5.

Third, note that the socket() constructor does *not* take a list of three items as its parameter. Instead, the parameter list is introduced by an asterisk, which means that the three elements of the socket_args list are passed as three separate parameters to the constructor. This is the opposite of what you need to do with the actual address returned, which is instead passed as a single unit into all of the socket routines that need it.

Better Living Through Paranoia

In certain high-security situations, people worry about trusting a hostname provided by an untrusted organization because there is nothing to stop you from creating a domain and pointing the hostnames inside it at the servers that actually belong to other organizations. For example, imagine that you provide a load-testing service, and that someone from example.com comes along and asks you to perform a murderously heavy test on their test.example.com server to see how their web server configuration holds up. The first thing you might ask yourself is whether they really own the host at test.example.com, or whether they have created that name in their domain but given it the IP address of the main web server of a competing organization so that your "test" in fact shuts their competition down for the afternoon.

But since it is common to have service-specific hostnames like gatech.edu point to the IP address of a real host like brahma2.gatech.edu, it can actually be rather tricky to determine if a reverse name mismatch indicates a problem. Ignoring the first element can be helpful, as can truncating both hostnames to the length of the shorter one—but the result might still be something that should be looked at by a human before making real access-control decisions based on the result!

But, to show you the sort of checking that might be attempted, you can take a look at Listing 4–2 for a possible sanity check that you might want to perform before starting the load test.

Listing 4–2. *Confirming a Forward Lookup with a Reverse Lookup*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 4 - forward_reverse.py
# Checking whether a hostname works both forward and backward.

import socket, sys

if len(sys.argv) != 2:
»   print >>sys.stderr, 'usage: forward_reverse.py <hostname>'
»   sys.exit(2)
hostname = sys.argv[1]

try:
»   infolist = socket.getaddrinfo(
»   »   hostname, 0, 0, socket.SOCK_STREAM, 0,
»   »   socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME,
»   »   )
except socket.gaierror, e:
»   print 'Forward name service failure:', e.args[1]
»   sys.exit(1)

info = infolist[0]  # choose the first, if there are several addresses
```

```
canonical = info[3]
socketname = info[4]
ip = socketname[0]

if not canonical:
»   print 'WARNING!  The IP address', ip, 'has no reverse name'
»   sys.exit(1)

print hostname, 'has IP address', ip
print ip, 'has the canonical hostname', canonical

# Lowercase for case-insensitive comparison, and chop off hostnames.

forward = hostname.lower().split('.')
reverse = canonical.lower().split('.')

if forward == reverse:
»   print 'Wow, the names agree completely!'
»   sys.exit(0)

# Truncate the domain names, which now look like ['www', 'mit', 'edu'],
# to the same length and compare.  Failing that, be willing to try a
# compare with the first element (the hostname?) lopped off if both of
# them are the same length.

length = min(len(forward), len(reverse))
if (forward[-length:] == reverse[-length:]
»   or (len(forward) == len(reverse)
»   »   and forward[-length+1:] == reverse[-length+1:]
»   »   and len(forward[-2]) > 2)):  # avoid thinking '.co.uk' means a match!
»   print 'The forward and reverse names have a lot in common'
else:
»   print 'WARNING!  The reverse name belongs to a different organization'
```

Here, we are not only telling `getaddrinfo()`to perform the “forward” lookup that resolves a hostname to an IP address, but also asking for the “reverse” lookup to discover what the actual owner of the IP address says that he or she has named that machine.

Using this script, you can see that some hosts have quite straightforward names that reverse to exactly the same string:

```
$ python forward_reverse.py smtp1.google.com
smtp1.google.com has IP address 216.239.44.95
216.239.44.95 has the canonical hostname smtp1.google.com
Wow, the names agree completely!
```

On the other hand, it is common for web site names that are designed to be short and pretty to actually be served by physical machines with longer names:

```
$ python forward_reverse.py mit.edu
mit.edu has IP address 18.9.22.69
18.9.22.69 has the canonical hostname WEB.MIT.EDU
The forward and reverse names have a lot in common
```

But very often a name is completely symbolic, and the site or services behind it are actually provided by machines in a completely different domain for perfectly legitimate reasons, but there is no way for our little script to know this:

```
$ python forward_reverse.py flickr.com
flickr.com has IP address 68.142.214.24
68.142.214.24 has the canonical hostname www.flickr.vip.mud.yahoo.com
WARNING! The reverse name belongs to a different organization
```

This means that unless you are writing code for a very specific situation in which you know that hostnames and their reverse names should strictly correspond, something like Listing 4–2 will be far too strict.

What, then, is the real usefulness of reverse lookups? The big reason is to have a second name to test against whatever lists of allowed and disallowed hosts your user might have configured. Of course, if the connection is an incoming one rather than an outgoing one, then the reverse name—which `getsockaddr()` will fetch for you if you provide the remote socket name—will be the only name you have to go on; forward names exist, of course, only when you are doing the connecting yourself based on a name that a user has configured or typed.

And here we conclude our discussion of how you should best do name resolution in your Python programs. But what if you need to go one level deeper—what if your application needs to speak to the name service infrastructure on its own for some reason? Then keep reading, and we will soon learn about DNS, which drives name resolution on IP networks!

A Sketch of How DNS Works

The Domain Name System, DNS, is a scheme by which millions of Internet hosts cooperate to answer the question of what hostnames resolve to what IP addresses. The DNS is behind the fact that you can type `python.org` into your web browser instead of always having to remember `82.94.164.162` for those of you on IPv4, or `2001:888:2000:d::a2` if you are already enjoying IPv6.

THE DNS PROTOCOL

Purpose: **Turn hostnames into IP addresses**

Standard: **RFC 1035 (1987) and subsequent**

Runs atop: **TCP/IP** and **UDP/IP**

Default port: **53**

Libraries: PyDNS, dnspython

Exceptions: See library documentation

The messages that computers send to perform this resolution form the “DNS Protocol,” which operates in a hierarchical fashion. If your local computer and nameserver cannot resolve a hostname because it is neither local to your organization nor has been seen recently enough to still be in the nameserver's cache, then the next step is to query one of the world's top-level nameservers to find out which machines are responsible for the domain you need to ask about. Once their IP addresses are ascertained, they can then be queried for the domain name itself.

We should first step back for a moment and see how this operation is usually set in motion.

For example, consider the domain name www.python.org. If your web browser needs to know this address, then the browser runs a call like getaddrinfo() to ask the operating system to resolve that name. Your system will know either that it is running a nameserver of its own, or that the network to which it is attached provides name service. Nameserver information these days is typically learned automatically through DHCP, whether in corporate offices, in schools, on wireless networks, or on home cable and DSL connections. In other cases, the DNS server IP addresses will have been configured by hand when a system administrator set up your machine. Either way, the DNS servers must typically be specified as IP addresses, since you obviously cannot use DNS itself to find them!

Sometimes people are unhappy with their ISP's DNS behavior or performance and choose to configure a third-party DNS server of their own choosing, like the servers at 8.8.8.8 and 8.8.4.4 run by Google. And in some rarer cases, the local DNS domain nameservers are known through some other set of names in use by the computer, like the WINS Windows naming service. But one way or another, a DNS server must be identified for name resolution to continue.

Checking DNS for the hostname is not actually the first thing that an operating system usually does when you make a call like getaddrinfo()—in fact, because making a DNS query can be time-consuming, it is often the very last choice! Depending on the hosts entry in your /etc/nsswitch.conf if you are on a POSIX box, or else depending on your Windows control panel settings, there might be one or several other places that the operating system looks first before turning to DNS. On my Ubuntu laptop, for example, the /etc/hosts file is checked first on every single hostname lookup; then a specialized protocol called multicast DNS is used, if possible; and only if that fails or is unavailable is full-blown DNS cranked up to answer the hostname query.

To continue our example, imagine that the name www.python.org has not, in fact, been recently enough queried to be in any local cache on the machine where you are running your web browser. In that case, the computer will look up the local DNS server and, typically, send it a DNS request packet over UDP.

Now the question is in the hands of a real DNS server! For the rest of this discussion, we will call it "your DNS server," in the sense of "the DNS server that is doing work for you"; but, of course, the server itself probably belongs to someone else, like your employer or your ISP or Google!

The first act of your DNS server will be to check its own cache of recently queried domain names to see if www.python.org has already been checked by some other machine served by the DNS server in the last few minutes or hours. If an entry is present and has not yet expired—and the owner of each domain name gets to choose its expiration timeout, because some organizations like to change IP addresses quickly if they need to, while others are happy to have old IP addresses linger for hours or days in the world's DNS caches—then it can be returned immediately. But let us imagine that it is morning and that you are the first person in your office or in the coffee shop to try talking to www.python.org today, and so the DNS server has to go find the hostname from scratch.

Your DNS server will now begin a recursive process of asking about www.python.org at the very top of the world's DNS server hierarchy: the "root-level" nameservers that know all of the top-level domains (TLDs) like .com, .org, .net, and all of the country domains, and know the groups of servers that are responsible for each. Nameserver software generally comes with the IP addresses of these top-level servers built in, to solve the bootstrapping problem of how you find any domain nameservers before you are actually connected to the domain name system! With this first UDP round-trip, your DNS server will learn (if it did not know already from another recent query) which servers keep the full index of .org domain.

Now a second DNS request will be made, this time to one of the .org servers, asking who on earth runs the python.org domain. You can find out what those top-level servers know about a domain by running the whois command-line program on a POSIX system, or use one of the many "whois" web pages online:

```
$ whois python.org
Domain Name:PYTHON.ORG
Created On:27-Mar-1995 05:00:00 UTC
Last Updated On:07-Sep-2006 20:50:54 UTC
```

```
Expiration Date:28-Mar-2016 05:00:00 UTC
...
Registrant Name:Python Software Foundation
...
Name Server:NS2.XS4ALL.NL
Name Server:NS.XS4ALL.NL
```

And that provides our answer! Wherever you are in the world, your DNS request for any hostname within `python.org` must be passed on to one of the two DNS servers named in that entry. Of course, when your DNS server makes this request to a top-level domain nameserver, it does not really get back only two names like those just given; instead, it is also given their IP addresses so that it can contact them directly without incurring another round of DNS lookup.

Your DNS server is now finished talking to both the root-level DNS server and the top-level `.org` DNS server, and can communicate directly with `NS2.XS4ALL.NL` or `NS.XS4ALL.NL` to ask about the `python.org` domain—and, in fact, it will usually try one of them and then fall back to trying the other if the first one is unavailable. This increases the chances of you getting an answer, but, of course, it can increase the amount of time that you sit staring at your web browser before the page can actually be displayed!

Depending on how `python.org` has its nameservers configured, the DNS server might require just one more query to get its answer, or it might take several if the organization is a large one with many departments and sub-departments that all run their own DNS servers to which requests need to be delegated. In this case, the `www.python.org` query can be answered directly by either of the two servers just named, and your DNS server can now return a UDP packet to your browser telling it which IP addresses belong to that hostname.

Note that this process required four separate network round-trips. Your machine made a request and got a response from your own DNS server, and in order to answer that request, your DNS server had to make a recursive query that consisted of three different round-trips to other servers. No wonder your browser sits there spinning when you enter a domain name for the first time!

Why Not to Use DNS

The foregoing explanation of a typical DNS query has, I hope, made clear that your operating system is doing a lot for you when you need a hostname looked up. For this reason, I am going to recommend that, unless you absolutely need to speak DNS for some quite particular reason, you always rely on `getaddrinfo()` or some other system-supported mechanism for resolving hostnames. Consider the benefits:

- The DNS is often not the only way that a system gets name information. If your application runs off and tries to use DNS on its own as its first choice for resolving a domain name, then users will notice that some computer names that work everywhere else on your system—in their browser, in file share names, and so forth—suddenly do not work when they use your application, because you are not deferring to mechanisms like WINS or `/etc/hosts` like the operating system itself does.

- The local machine probably has a cache of recently queried domain names that might already know about the host whose IP address you need. If you try speaking DNS yourself to answer your query, you will be duplicating work that has already been done.

- The system on which your Python script is running already knows about the local domain nameservers, thanks either to manual intervention by your system administrator or a network configuration protocol like DHCP in your office, home, or coffee shop. To crank up DNS right inside your Python program, you will have to learn how to query your particular operating system for this information—an operating-system-specific action that we will not be covering in this book.
- If you do not use the local DNS server, then you will not be able to benefit from its own cache that would prevent your application and other applications running on the same network from repeating requests about a hostname that is in frequent use at your location.
- From time to time, adjustments are made to the world DNS infrastructure, and operating system libraries and daemons are gradually updated to accommodate this. If your program makes raw DNS calls of its own, then you will have to follow these changes yourself and make sure that your code stays up-to-date with the latest changes in TLD server IP addresses, conventions involving internationalization, and tweaks to the DNS protocol itself.

Finally, note that Python does not come with any DNS facilities built into the Standard Library. If you are going to talk DNS using Python, then you must choose and learn a third-party library for doing so.

Why to Use DNS

There is, however, a solid and legitimate reason to make a DNS call from Python: because you are a mail server, or at the very least a client trying to send mail directly to your recipients without needing to run a local mail relay, and you want to look up the `MX` records associated with a domain so that you can find the correct mail server for your friends at `@example.com`.

So we are going to go ahead and take a look at one of the third-party DNS libraries for Python as we bring this chapter to its close. There are at least two good ones available for Python at the moment. They are available for quick installation into a virtual environment if you want to try them out. (See Chapter 1 to remember how to use `virtualenv` and `pip`.)

We will focus on the slightly more popular distribution, `pydns`, which descends from a DNS module first written by Guido van Rossum, which at least gives it a glow of historical legitimacy. It makes a `DNS` package available for you to import. Its competitor, the `dnspython` distribution, creates a lower-case `dns` package instead, just so you can keep things straight! Both distributions have seen updates within the past year, as of this writing—in fact, as I type this in September 2010, I can see that both packages were updated within a few days of each other back in January 2010.

Note that neither project provides code that knows how to "start from scratch" and begin a query with a search of the Internet root domain nameservers! Instead, each library uses its own tricks to find out what domain nameservers your Windows or POSIX operating system is currently using, and then asks those servers to go do recursive queries on its behalf. So not a single piece of code in this chapter avoids needing to have a correctly configured host which an administrator or network configuration service has already configured with working nameservers.

Since both are on the Python Package Index, you can install and try one of them out like this:

```
$ pip install pydns
```

Your Python interpreter will then gain the ability to run our first DNS program listing, shown in Listing 4–3. Neither package seems to have any real documentation, so you will have to start with what is shown here and extrapolate by reading whatever example code you can find on the Web.

Listing 4–3. *A Simple DNS Query Doing Its Own Recursion*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 4 - dns_basic.py
# Basic DNS query

import sys, DNS

if len(sys.argv) != 2:
»   print >>sys.stderr, 'usage: dns_basic.py <hostname>'
»   sys.exit(2)

DNS.DiscoverNameServers()
request = DNS.Request()
for qt in DNS.Type.A, DNS.Type.AAAA, DNS.Type.CNAME, DNS.Type.MX, DNS.Type.NS:
»   reply = request.req(name=sys.argv[1], qtype=qt)
»   for answer in reply.answers:
»   »   print answer['name'], answer['classstr'], answer['typename'], \
»   »   »   repr(answer['data'])
```

Running this against `python.org` will immediately teach us several things about DNS:

```
$ python dns_basic.py python.org
python.org IN A '82.94.164.162'
python.org IN AAAA ' \x01\x08\x88 \x00\x00\r\x00\x00\x00\x00\x00\x00\x00\xa2'
python.org IN MX (50, 'mail.python.org')
python.org IN NS 'ns2.xs4all.nl'
python.org IN NS 'ns.xs4all.nl'
```

As you can see from the program, each "answer" in the reply that has been returned is represented by a dictionary in `pydns`, and we are here grabbing a few of its most important keys and printing them out. In order, the keys that get printed on each line are as follows:

- The name that we looked up.
- The "class," which in all queries you are likely to see is `IN`, meaning it is a question about Internet addresses.
- The "type" of record; some common ones are `A` for an IPv4 address, `AAAA` for an IPv6 address, `NS` for a record that lists a nameserver, and `MX` for a statement about what mail server should be used for a domain.
- Finally, the "data" provides the information for which the record type was essentially a promise: the address, or data, or hostname associated with the name that we asked about.

In the query just quoted, we learn three things about the `python.org` domain. First, the `A` record tells us that if we want to connect to an actual `python.org` machine—to make an HTTP connection, or start an SSH session, or to do anything else because the user has supplied `python.org` as the machine he or she wants to connect to—then we should direct our packets at IP address `82.94.164.162`. Second, the `NS` records tell us that if we want the names of any hosts beneath `python.org`, then we should ask the two nameservers `ns2.xs4all.nl` and `ns.xs4all.nl` to resolve those names for us. Finally, if we want to send e-mail to someone at the e-mail domain `@python.org`, then we will need to go look up the hostname `mail.python.org` and connect to its SMTP port.

There is also a record type `CNAME`, which indicates that the hostname you have queried about is actually just an alias for another hostname—that you then have to go and look up separately! Because it

often requires two round-trips, this record type is unpopular these days, but you still might run across it occasionally.

That `MX` record is crucial, by the way, and is something that newcomers to network programming often get confused! Sending e-mail to a domain is a completely different act from trying to make an HTTP or SSH connection to a domain; if you want to send e-mail to someone `@python.org`, then do *not* try making an SMTP connection to the *host* named `python.org`! Always rely on `MX` records to point you to your destination, if they exist; try making an SMTP connection to an `A` record for the domain named in an e-mail address only if there are no `MX` records returned for that domain name.

Resolving Mail Domains

I mentioned previously that resolving an e-mail domain is a very legitimate use of raw DNS in most Python programs. The rules for doing this resolution are specified most recently in RFC 5321. They are, briefly, that if `MX` records exist, then you must try to contact those SMTP servers, and return an error to the user (or put the message on a re-try queue) if none of them will accept the message. If instead no `MX` records exist, but an `A` or `AAAA` record is provided for the domain, then you are allowed to try an SMTP connection to that address. If neither record exists, but a `CNAME` is specified, then the domain name it provides should be searched for `MX` or `A` records using the same rules.

Listing 4–4 shows how you might implement this algorithm. By doing a series of DNS queries, it works its way through the possible destinations, printing out its decisions as it goes. By adjusting a routine like this to return addresses rather than just printing them out, you could power a Python mail dispatcher that needed to deliver e-mail to remote hosts.

Listing 4–4. *Resolving an E-mail Domain Name*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 4 - dns_mx.py
# Looking up a mail domain - the part of an email address after the `@`

import sys, DNS

if len(sys.argv) != 2:
    print >>sys.stderr, 'usage: dns_basic.py <hostname>'
    sys.exit(2)

def resolve_hostname(hostname, indent=0):
    """Print an A or AAAA record for `hostname`; follow CNAMEs if necessary."""
    indent = indent + 4
    istr = ' ' * indent
    request = DNS.Request()
    reply = request.req(name=sys.argv[1], qtype=DNS.Type.A)
    if reply.answers:
        for answer in reply.answers:
            print istr, 'Hostname', hostname, '= A', answer['data']
        return
    reply = request.req(name=sys.argv[1], qtype=DNS.Type.AAAA)
    if reply.answers:
        for answer in reply.answers:
            print istr, 'Hostname', hostname, '= AAAA', answer['data']
        return
    reply = request.req(name=sys.argv[1], qtype=DNS.Type.CNAME)
    if reply.answers:
```

```
»   »   cname = reply.answers[0]['data']
»   »   print istr, 'Hostname', hostname, 'is an alias for', cname
»   »   resolve_hostname(cname, indent)
»   »   return
»   print istr, 'ERROR: no records for', hostname

def resolve_email_domain(domain):
»   """Print mail server IP addresses for an email address @ `domain`."""
»   request = DNS.Request()
»   reply = request.req(name=sys.argv[1], qtype=DNS.Type.MX)
»   if reply.answers:
»   »   print 'The domain %r has explicit MX records!' % (domain,)
»   »   print 'Try the servers in this order:'
»   »   datalist = [ answer['data'] for answer in reply.answers ]
»   »   datalist.sort()  # lower-priority integers go first
»   »   for data in datalist:
»   »   »   priority = data[0]
»   »   »   hostname = data[1]
»   »   »   print 'Priority:', priority, '  Hostname:', hostname
»   »   »   resolve_hostname(hostname)
»   else:
»   »   print 'Drat, this domain has no explicit MX records'
»   »   print 'We will have to try resolving it as an A, AAAA, or CNAME'
»   »   resolve_hostname(domain)

DNS.DiscoverNameServers()
resolve_email_domain(sys.argv[1])
```

Of course, the implementation of `resolve_hostname()` shown here is rather fragile, since it should really have a dynamic preference between `A` and `AAAA` records based on whether the current host is connected to an IPv4 or to an IPv6 network. In fact, it is likely that our friend `getsockaddr()` should really be deferred to here instead of trying to resolve the hostname ourselves! But since Listing 4–4 is designed to show off how the DNS works, I thought I might as well follow through with the logic using pure DNS so that you could see how the queries are resolved.

A real mail server implementation, instead of printing out the mail server addresses, would obviously attempt to deliver mail to them instead, and stop once the first success was achieved. (If it kept going through the server list after the success, then several copies of the e-mail would be generated, one for each server to which it was delivered successfully!) But this simple script gives us a good idea of the process. We can see that `python.org` at the moment has but a single mail server IP address:

```
$ python dns_mx.py python.org
The domain 'python.org' has explicit MX records!
Try the servers in this order:
Priority: 50   Hostname: mail.python.org
»    Hostname mail.python.org = A 82.94.164.162
```

Whether that IP belongs to one machine, or is shared by a cluster of hosts, is, of course, something that we cannot easily see from outside. Other organizations are more aggressive in giving incoming e-mails several places to land; the IANA currently has no fewer than eight e-mail servers:

```
$ python dns_mx.py iana.org
The domain 'iana.org' has explicit MX records!
Try the servers in this order:
Priority: 10   Hostname: pechora1.icann.org
»    Hostname pechora1.icann.org = A 192.0.43.8
```

```
Priority: 10   Hostname: pechora2.icann.org
»    Hostname pechora2.icann.org = A 192.0.43.8
...
Priority: 10   Hostname: pechora8.icann.org
»    Hostname pechora8.icann.org = A 192.0.43.8
```

By trying this script against many different domains, you will be able to see how both big and small organizations arrange for incoming e-mails to be routed to IP addresses.

Zeroconf and Dynamic DNS

There are two last technologies that you are perhaps not likely to implement yourself, but that deserve a quick mention because they allow machines to find each other when they lack permanent and stable IP addresses.

The *Zeroconf* standard combines three techniques so that computers thrown on to a network segment with each other can discover each other's presence and names without a network administrator needing to install and configure a DHCP and DNS server. Apple computers use this technology extensively to find adjacent machines and printers, Linux machines often run an `avahi` service that implements the protocol, and there is an old `pyzeroconf` project that offers a complete Python implementation of the protocol suite. One of the technologies included in Zeroconf is “multicast DNS” (mDNS), which allows all of the machines on the local network to answer when another machine needs to look up a hostname.

Dynamic DNS services are Internet sites built to serve users whose machines are regularly changing their IP address—perhaps because the address assigned by their ISP is not stable but is pulled from a pool of free addresses with every reconnect. By offering an API through which the user can offer her username, password, and new IP address, the DDNS service can update its database and point the user's domain name at the new IP. This technology was pioneered by the `dyndns.com` site, and it absolves the user of the need to rent and operate his or her own DNS server if he or she has only a few domain names to maintain. There appears to be a `dyndnsc` project on the Package Index that offers a client that can communicate with DDNS services.

Summary

Python programs often have to turn hostnames into socket addresses to which they can actually make connections.

Most hostname lookup should occur through the `getsockaddr()` function in the `socket` module, since its intelligence is usually supplied by your operating system and it will know not only how to look up domain names, but also what flavor of address the local IP stack is configured to support.

Old IPv4 addresses are still the most prevalent on the Internet, but IPv6 is becoming more and more common. By deferring all hostname and port name lookup to `getsockaddr()`, your Python program can treat addresses as opaque strings and not have to worry about parsing or interpreting them.

Behind most name resolution is the DNS, a worldwide distributed database that forwards domain name queries directly to the servers of the organization that owns a domain. While not often used directly from Python, it can be very helpful in determining where to direct e-mail based on the e-mail domain named after the `@` sign in an e-mail address.

CHAPTER 5

■ ■ ■

Network Data and Network Errors

The first four chapters have given us a foundation: we have learned how hosts are named on an IP network, and we understand how to set up and tear down both TCP streams and UDP datagram connections between those hosts.

But what data should we then send across those lengths? How should it be encoded and formatted? For what kinds of errors will our Python programs need to be prepared?

These questions are all relevant regardless of whether we are using streams or datagrams. We will look at the basic answers in this chapter, and learn how to use sockets responsibly so that our data arrives intact.

Text and Encodings

If you were watching for it as you read the first few chapters, you may have caught me using two different terms for the same concept. Those terms were *byte* and *octet*, and by both words I always mean an 8-bit number—an ordered sequence of eight digits, that are each either a one or a zero. They are the fundamental units of data on modern computing systems, used both to represent raw binary numbers and to stand for characters or symbols. The binary number 1010000, for example, usually stands for either the number 80 or the letter P:

```
>>> 0b1010000
80
>>> chr(0b1010000)
'P'
```

The reason that the Internet RFCs are so inveterate in their use of the term “octet” instead of “byte” is that the earliest of RFCs date from a very ancient era in which bytes could be one of several different lengths—byte sizes from as little as 5 to as many as 16 bits were used on various systems. So the term “octet,” meaning a “group of eight things,” is always used in the standards so that their meaning is unambiguous.

Four bits offer a mere sixteen values, which does not come close to even fitting our alphabet. But eight bits—the next-higher multiple of two—proved more than enough to fit both the upper and lower cases of our alphabet, all the digits, lots of punctuation, and 32 control codes, and it still left a whole half of the possible range of values empty. The problem is that many rival systems exist for the specific mapping used to turn characters into bytes, and the differences can cause problems unless both ends of your network connection use the same rules.

The use of ASCII for the basic English letters and numbers is nearly universal among network protocols these days. But when you begin to use more interesting characters, you have to be careful. In Python you should always represent a meaningful string of text with a “Unicode string” that is denoted with a leading `u`, like this:

```
>>> elvish = u'Namárië!'
```

But you cannot put such strings directly on a network connection without specifying which rival system of encoding you want to use to mix your characters down to bytes. A very popular system is UTF-8, because normal characters are represented by the same codes as in ASCII, and longer sequences of bytes are necessary only for international characters:

```
>>> elvish.encode('utf-8')
'Nam\xc3\xa1ri\xc3\xab!'
```

You can see, for example, that UTF-8 represented the letter *ë* by a pair of bytes with hex values C3 and AB.

Be very sure, by the way, that you understand what it means when Python prints out a normal string like the one just given. The letters strung between quotation characters with no leading `u` do *not* inherently represent letters; they do not inherently represent anything until your program decides to do something with them. They are just bytes, and Python is willing to store them for you without having the foggiest idea what they mean.

Other encodings are available in Python—the Standard Library documentation for the `codecs` package lists them all. They each represent a full system for reducing symbols to bytes. Here are a few examples of the byte strings produced when you try encoding the same word in different ways; because each successive example has less in common with ASCII, you will see that Python's choice to use ASCII to represent the bytes in strings makes less and less sense:

```
>>> elvish.encode('utf-16')
'\xff\xfeN\x00a\x00m\x00\xe1\x00r\x00i\x00\xeb\x00!\x00'
>>> elvish.encode('cp1252')
'Nam\xe1ri\xeb!'
>>> elvish.encode('idna')
'xn--namri!-rta6f'
>>> elvish.encode('cp500')
'\xd5\x81\x94E\x99\x89SO'
```

You might be surprised that my first example was the encoding UTF-16, since at first glance it seems to have created a far greater mess than the encodings that follow. But if you look closely, you will see that it is simply using two bytes—sixteen bits—for each character, so that most of the characters are simply a null character `\x00` followed by the plain ASCII character that belongs in the string. (Note that the string also begins with a special sequence `\xff\xfe` that designates the byte order in use; see the next section for more about this concept.)

On the receiving end of such a string, simply take the byte string and call its `decode()` method with the name of the codec that was used to encode it:

```
>>> print '\xd5\x81\x94E\x99\x89SO'.decode('cp500')
Namárië!
```

These two steps—encoding to a byte string, and then decoding again on the receiving end—are essential if you are sending real text across the network and want it to arrive intact. Some of the protocols that we will learn about later in this book handle encodings for you (see, for example, the description of HTTP in Chapter 9), but if you are going to write byte strings to raw sockets, then you will not be able to avoid tackling the issue yourself.

Of course, many encodings do not support enough characters to encode all of the symbols in certain pieces of text. The old-fashioned 7-bit ASCII encoding, for example, simply cannot represent the string we have been working with:

```
>>> elvish.encode('ascii')
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe1' in position 3: ordinal↩
 not in range(128)
```

Note that some encodings have the property that every character they are able to encode will be represented by the same number of bytes; ASCII uses one byte for every character, for example, and UTF-32 uses four. If you use one of these encodings, then you can both determine the number of characters in a string by a simple examination of the number of bytes it contains, and jump to character *n* of the string very efficiently. (Note that UTF-16 does not have this property, since it uses 16 bits for some characters and 32 bits for others.)

Some encodings also add prefix characters that are not part of the string, but help the decoder detect the byte ordering that was used (byte order is discussed in the next section)—thus the `\xff\xfe` prefix that Python's UTF-16 encoder added to the beginning of our string. Read the `codecs` package documentation and, if necessary, the specifications for particular encodings to learn more about the actions they perform when turning your stream of symbols into bytes.

Note that it is dangerous to decode a partially received message if you are using an encoding that encodes some characters using multiple bytes, since one of those characters might have been split between the part of the message that you have already received and the packets that have not yet arrived. See the section later in this chapter on "Framing" for some approaches to this issue.

Network Byte Order

If all you ever want to send across the network is text, then encoding and framing (which we tackle in the next section) will be your only worries.

But sometimes you might want to represent your data in a more compact format than text makes possible. Or you might be writing Python code to interface with a service that has already made the choice to use raw binary data. In either case, you will probably have to start worrying about a new issue: network byte order.

To understand the issue of byte order, consider the process of sending an integer over the network. To be specific, think about the integer 4253.

Many protocols, of course, will simply transmit this integer as the string `'4253'`—that is, as four distinct characters. The four digits will require at least four bytes to transmit, at least in any common text encoding. And using decimal digits will also involve some computational expense: since numbers are not stored inside computers in base 10, it will take repeated division—with inspection of the remainder—to determine that this number is in fact made of 4 thousands, plus 2 hundreds, plus 5 tens, plus 3 left over. And when the four-digit string `'4253'` is received, repeated addition and multiplication by powers of ten will be necessary to put the text back together into a number.

Despite its verbosity, the technique of using plain text for numbers may actually be the most popular on the Internet today. Every time you fetch a web page, for example, the HTTP protocol expresses the `Content-Length` of the result using a string of decimal digits just like `'4253'`. Both the web server and client do the decimal conversion without a second thought, despite the bit of expense. Much of the story of the last 20 years in networking, in fact, has been the replacement of dense binary formats with protocols that are simple, obvious, and human-readable—even if computationally expensive compared to their predecessors.

(Of course, multiplication and division are also cheaper on modern processors than back when binary formats were more common—not only because processors have experienced a vast increase in speed, but because their designers have become much more clever about implementing integer math, so that the same operation requires far fewer cycles today than on the processors of, say, the early 1980s.)

In any case, the string `'4253'` is not how your computer represents this number as an integer variable in Python. Instead it will store it as a binary number, using the bits of several successive bytes to represent the one's place, two's place, four's place, and so forth of a single large number. We can glimpse the way that the integer is stored by using the `hex()` built-in function at the Python prompt:

```
>>> hex(4253)
'0x109d'
```

Each hex digit corresponds to four bits, so each pair of hex digits represents a byte of data. Instead of being stored as four decimal digits `4`, `4`, `2`, and `3` with the first `4` being the "most significant" digit (since tweaking its value would throw the number off by a thousand) and `3` being its least significant digit, the number is stored as a most significant byte `0x10` and a least significant byte `0x9d`, adjacent to one another in memory.

But in which order should these two bytes appear? Here we reach a great difference between computers. While they will all agree that the bytes in memory have an order, and they will all store a string like `Content-Length: 4253` in exactly that order starting with `C` and ending with `3`, they do not share a single idea about the order in which the bytes of a binary number should be stored.

Some computers are "big-endian" (for example, older SPARC processors) and put the most significant byte first, just like we do when writing decimal digits; others (like the nearly ubiquitous x86 architecture) are "little-endian" and put the least significant byte first.

For an entertaining historical perspective on the issue, be sure to read Danny Cohen's paper IEN-137, "On Holy Wars and a Plea for Peace," which introduced the words "big-endian" and "little-endian" in a parody of Jonathan Swift: `www.ietf.org/rfc/ien/ien137.txt.`

Python makes it very easy to see the difference between the two endiannesses. Simply use the `struct` module, which provides a variety of operations for converting data to and from popular binary formats. Here is the number 4253 represented first in a little-endian format and then in a big-endian order:

```
>>> import struct
>>> struct.pack('<i', 4253)
'\x9d\x10\x00\x00'
>>> struct.pack('>i', 4253)
'\x00\x00\x10\x9d'
```

We here used the code `i`, which uses four bytes to store an integer, so the two upper bytes are zero for a small number like 4253. You can think of the `struct` codes for these two orders as little arrows pointing toward the least significant end of a string of bytes, if that helps you remember which one to use. See the `struct` module documentation in the Standard Library for the full array of data formats that it supports. It also supports an `unpack()` operation, which converts the binary data back to Python numbers:

```
>>> struct.unpack('>i', '\x00\x00\x10\x9d')
(4253,)
```

If the big-endian format makes more sense to you intuitively, then you may be pleased to learn that it "won" the contest of which endianness would become the standard for network data. Therefore the `struct` module provides another symbol, `'!'`, which means the same thing as `'>'` when used in `pack()` and `unpack()` but says to other programmers (and, of course, to yourself as you read the code later), "I am packing this data so that I can send it over the network."

In summary, here is my advice for preparing binary data for transmission across a network socket:

- Use the `struct` module to produce binary data for transmission on the network, and to unpack it upon arrival.
- Select network byte order with the `'!'` prefix if the data format is up to you.
- If someone else has designed the protocol and specified little-endian, then you will have to use `'<'` instead.
- Always test `struct` to see how it lays out your data compared to the specification for the protocol you are speaking; note that `'x'` characters in the packing format string can be used to insert padding bytes.

You might see older Python code use a cadre of awkwardly named functions from the `socket` module in order to turn integers into byte strings in network order. These functions have names like `ntohl()` and `htons()`, and correspond to functions of the same name in the POSIX networking library—

which also supplies calls like `socket()` and `bind()`. I suggest that you ignore these awkward functions, and use the `struct` module instead; it is more flexible, more general, and produces more readable code.

Framing and Quoting

If you are using UDP datagrams for communication, then the protocol itself takes the trouble to deliver your data in discrete and identifiable chunks—and you have to reorder and re-transmit them yourself if anything goes wrong on the network, as outlined in Chapter 2.

But if you have made the far more common option of using a TCP stream for communication, then you will face the issue of *framing*—of how to delimit your messages so that the receiver can tell where one message ends and the next begins. Since the data you supply to `sendall()` might be broken up into several packets, the program that receives your message might have to make several `recv()` calls before your whole message has been read.

The issue of framing asks the question: when is it safe for the receiver to finally stop calling `recv()` and respond to your message?

As you might imagine, there are several approaches.

First, there is a pattern that can be used by extremely simple network protocols that involve only the delivery of data—no response is expected, so there never has to come a time when the receiver decides "Enough!" and turns around to send a response. In this case, the sender can loop until all of the outgoing data has been passed to `sendall()` and then `close()` the socket. The receiver need only call `recv()` repeatedly until the call finally returns an empty string, indicating that the sender has finally closed the socket. You can see this pattern in Listing 5–1.

Listing 5–1. *Sending a Single Stream of Data*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 5 - streamer.py
# Client that sends data then closes the socket, not expecting a reply.

import socket, sys
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = sys.argv.pop() if len(sys.argv) == 3 else '127.0.0.1'
PORT = 1060

if sys.argv[1:] == ['server']:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    print 'Listening at', s.getsockname()
    sc, sockname = s.accept()
    print 'Accepted connection from', sockname
    sc.shutdown(socket.SHUT_WR)
    message = ''
    while True:
        more = sc.recv(8192)  # arbitrary value of 8k
        if not more:  # socket has closed when recv() returns ''
            break
        message += more
    print 'Done receiving the message; it says:'
    print message
    sc.close()
```

```
    s.close()

elif sys.argv[1:] == ['client']:
    s.connect((HOST, PORT))
    s.shutdown(socket.SHUT_RD)
    s.sendall('Beautiful is better than ugly.\n')
    s.sendall('Explicit is better than implicit.\n')
    s.sendall('Simple is better than complex.\n')
    s.close()

else:
    print >>sys.stderr, 'usage: streamer.py server|client [host]'
```

If you run this script as a server and then, at another command prompt, run the client version, you will see that all of the client's data makes it intact to the server, with the end-of-file event generated by the client closing the socket serving as the only framing that is necessary:

```
$ python streamer.py server
Listening at ('127.0.0.1', 1060)
Accepted connection from ('127.0.0.1', 52039)
Done receiving the message; it says:
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

Note the nicety that, since this socket is not intended to receive any data, the client and server both go ahead and shut down communication in the other direction. This prevents any accidental use of the socket in the other direction—use that could eventually queue up enough unread data to produce deadlock, as we saw in Listing 3-2. It is really only necessary for either the client or server to call `shutdown()` on the socket; it is redundant for both of them to do so. But since you someday might be programming only one end of such a connection, I thought you might want to see how the shutdown looks from both directions.

A second pattern is a variant on the first: streaming in both directions. The socket is initially left open in both directions. First, data is streamed in one direction—exactly as shown in Listing 5–1—and then that direction alone is shut down. Second, data is then streamed in the other direction, and the socket is finally closed. Again, Listing 3-2 provides an important warning: always finish the data transfer in one direction before turning around to stream data back in the other, or you could produce a client and server that are deadlocked.

A third pattern, which we have already seen, is to use fixed-length messages, as illustrated in Listing 3-1. You can use the Python `sendall()` method to keep sending parts of a string until the whole thing has been transmitted, and then use a `recv()` loop of our own devising to make sure that you receive the whole message:

```
def recvall(sock, length):
    data = ''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('socket closed %d bytes into a %d-byte message'
                           % (len(data), length))
        data += more
    return data
```

Fixed-length messages are a bit rare since so little data these days seems to fit within static boundaries, but when transmitting binary data in particular, you might find it a good fit for certain situations.

A fourth pattern is to somehow delimit your messages with special characters. The receiver would wait in a `recv()` loop like the one just cited, but wait until the `reply` string it was accumulating finally contained the delimiter indicating the end-of-message. If the bytes or characters in the message are guaranteed to fall within some limited range, then the obvious choice is to end each message with a symbol chosen from outside that range. If you were sending ASCII strings, for example, you might choose the null character `'\0'` as the delimiter.

If instead the message can include arbitrary data, then using a delimiter is a problem: what if the character you are trying to use as the delimiter turns up as part of the data? The answer, of course, is quoting, just like having to represent a single-quote character as `\'` in the middle of a Python string that is itself delimited by single-quote characters:

```
'All\'s well that ends well.'
```

I recommend using a delimiter scheme only where your message alphabet is constrained; it is too much trouble if you have to handle arbitrary data. For one thing, your test for whether the delimiter has arrived now has to make sure that you are not confusing a quoted delimiter for a real one that actually ends the message. A second complexity is that you then have to make a pass over the message to remove the quote characters that were protecting literal occurrences of the delimiter. Finally, it means that message length cannot be measured until you have performed decoding—a message of length 400 could be 400 symbols long, or it could be 200 instances of the delimiter accompanied by the quoting character, or anything in between.

A fifth pattern is to prefix each message with its length. This is a very popular choice for high-performance protocols since blocks of binary data can be sent verbatim without having to be analyzed, quoted, or interpolated. Of course, the length itself has to be framed using one of the techniques given previously—often it is simply a fixed-width binary integer, or else a variable-length decimal string followed by a delimiter. But either way, once the length has been read and decoded, the receiver can enter a loop and call `recv()` repeatedly until the whole message has arrived. The loop can look exactly like the one in Listing 3-1, but with a length variable in place of the number 16.

Finally, what if you want the simplicity and efficiency of this fifth pattern but you do not know ahead of time how long each message will be—perhaps because the sender is himself reading data from a source whose length he cannot predict? In such cases, do you have to abandon elegance and slog through the data looking for delimiters?

Unknown lengths are no problem if you use a final, and sixth, pattern. Instead of sending just one, try sending several blocks of data that are each prefixed with their length. This means that as each chunk of new information becomes available to the sender, it can be labeled with its length and placed on the outgoing stream. When the end finally arrives, the sender can emit an agreed-upon signal—perhaps a length field giving the number zero—that tells the receiver that the series of blocks is complete.

A very simple example of this idea is shown in Listing 5–2. Like the previous listing, this sends data in only one direction—from the client to the server—but the data structure is much more interesting. Each message is prefixed with a 4-byte length; in a struct, `'I'` means a 32-bit unsigned integer, meaning that these messages can be up to 4GB in length. A series of three such messages is sent to the server, followed by a zero-length message—which is essentially just a length field with zeros inside and then no message data after it—to signal that the series of blocks is over.

Listing 5–2. *Sending Blocks of Data*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 5 - blocks.py
# Sending data one block at a time.

import socket, struct, sys
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

HOST = sys.argv.pop() if len(sys.argv) == 3 else '127.0.0.1'
PORT = 1060
format = struct.Struct('!I')  # for messages up to 2**32 - 1 in length

def recvall(sock, length):
    data = ''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('socket closed %d bytes into a %d-byte message'
                           % (len(data), length))
        data += more
    return data

def get(sock):
    lendata = recvall(sock, format.size)
    (length,) = format.unpack(lendata)
    return recvall(sock, length)

def put(sock, message):
    sock.send(format.pack(len(message)) + message)

if sys.argv[1:] == ['server']:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    print 'Listening at', s.getsockname()
    sc, sockname = s.accept()
    print 'Accepted connection from', sockname
    sc.shutdown(socket.SHUT_WR)
    while True:
        message = get(sc)
        if not message:
            break
        print 'Message says:', repr(message)
    sc.close()
    s.close()

elif sys.argv[1:] == ['client']:
    s.connect((HOST, PORT))
    s.shutdown(socket.SHUT_RD)
    put(s, 'Beautiful is better than ugly.')
    put(s, 'Explicit is better than implicit.')
    put(s, 'Simple is better than complex.')
    put(s, '')
    s.close()

else:
    print >>sys.stderr, 'usage: streamer.py server|client [host]'
```

Note how careful we have to be! Even though four bytes of length is such a tiny amount of data that we cannot imagine `recv()` not returning it all at once, our code is still correct only if we carefully wrap

recv() in a loop that—just in case—will keep demanding more data until all four bytes have arrived. This is the kind of caution that will serve you well when writing network code. It is also the kind of fiddly little detail that makes most people glad that they can deal just with higher-level protocols, and not have to learn to talk with sockets in the first place!

So those are six good options for dividing up an unending stream of data into digestible chunks so that clients and servers know when a message is complete and they can turn around and respond. Note that many modern protocols mix them together, and that you are free to do the same thing.

A good example is the HTTP protocol, which we will learn more about in Part 2 of this book. It uses a delimiter—the blank line '\r\n\r\n'—to signal when its headers are complete. Because the headers are text, line endings can safely be treated as special characters. But since the actual payload can be pure binary data, like an image or compressed file, the Content-Length provided in the headers is used to determine how much more data to read off of the socket. Thus HTTP mixes the fourth and fifth patterns we have looked at here. In fact, it can also use our sixth option: if a server is streaming a response whose length it cannot predict, then it can use a "chunked encoding," which sends several blocks that are each prefixed with their length. A zero length marks the end of the transmission, just as it does in Listing 5–2.

Pickles and Self-Delimiting Formats

Note that some kinds of data that you might send across the network already include some form of delimiting built-in. If you are transmitting such data, then you might not have to impose your own framing atop what the data is already doing.

Consider Python "pickles," for example, the native form of serialization that comes with the Standard Library. Using a quirky mix of text commands and data, a pickle stores the contents of a Python data structure so that you can reconstruct it later or on a different machine:

```
>>> import pickle
>>> pickle.dumps([5, 6, 7])
'(lp0\nI5\naI6\naI7\na.'
```

The interesting thing about the format is the '.' character that you see at the end of the foregoing string—it is the format's way of marking the end of a pickle. Upon encountering it, the loader can stop and return the value without reading any further. Thus we can take the foregoing pickle, stick some ugly data on the end, and see that loads() will completely ignore it and give us our original list back:

```
>>> pickle.loads('(lp0\nI5\naI6\naI7\na.UjJGdVpHRnNaZz09')
[5, 6, 7]
```

Of course, using loads() this way is not useful for network data, since it does not tell us how many bytes it processed in order to reload the pickle; we still do not know how much of our string is pickle data. But if we switch to reading from a file and using the pickle load() function, then the file pointer will be left right at the end of the pickle data, and we can start reading from there if we want to read what came after the pickle:

```
>>> from StringIO import StringIO
>>> f = StringIO('(lp0\nI5\naI6\naI7\na.UjJGdVpHRnNaZz09')
>>> pickle.load(f)
[5, 6, 7]
>>> f.pos
18
>>> f.read()
'UjJGdVpHRnNaZz09'
```

Alternately, we could create a protocol that just consisted of sending pickles back and forth between two Python programs. Note that we would not need the kind of loop that we put into the recvall()

function in Listing 5–2, because the pickle library knows all about reading from files and how it might have to do repeated reads until an entire pickle has been read. Remember to use the makefile() socket method—which was discussed in Chapter 3—if you want to wrap a socket in a Python file object for consumption by a routine like the pickle load() function.

Note that there are many subtleties involved in pickling large data structures, especially if they contain Python objects beyond simple built-in types like integers, strings, lists, and dictionaries. See the pickle module documentation for more details.

XML, JSON, Etc.

If your protocol needs to be usable from other programming languages—or if you simply prefer universal standards to formats specific to Python—then the JSON and XML data formats are each a popular choice. Note that neither of these formats supports framing, so you will have to first figure out how to extract a complete string of text from over the network before you can then process it.

JSON is among the best choices available today for sending data between different computer languages. Since Python 2.6, it has been included in the Standard Library as a module named json; for earlier Python versions, simply install the popular simplejson distribution. Either way, you will have available a universal technique for serializing simple data structures:

```
>>> try:
...     import json
... except ImportError:
...     import simplejson as json
...
>>> json.dumps([ 51, u'Namárië!' ])
'[51, "Nam\\u00e1ri\\u00eb!"]'
>>> json.loads('{"name": "Lancelot", "quest": "Grail"}')
{u'quest': u'Grail', u'name': u'Lancelot'}
```

Note that the protocol fully supports Unicode strings—using the popular UTF-8 as its default encoding—and that it supports strings of actual characters, not Python-style strings of bytes, as its basic type. For more information about JSON, see the discussion in Chapter 18 about JSON-RPC; that chapter talks in greater detail about the Python data types that the JSON format supports, and also has some hints about getting your data ready for serialization.

It does, however, have a weakness: a vast omission in the JSON standard is that it provides absolutely no provision for cleanly passing binary data like images or arbitrary documents. There exists a kindred format named BSON—the "B" is for "binary"—that supports additional types including raw binary strings. In return it sacrifices human readability, substituting raw binary octets and length fields for the friendly braces and quotation marks of JSON.

The XML format is better for documents, since its basic structure is to take strings and mark them up by wrapping them in angle-bracketed elements. In Chapter 10, we will take an extensive look at the various options available in Python for processing documents written in XML and related formats. But for now, simply keep in mind that you do not have to limit your use of XML to when you are actually using the HTTP protocol; there might be a circumstance when you need markup in text and you find XML useful in conjunction with some other protocol.

Among many other formats that developers might want to consider are Google Protocol Buffers, which are a bit different than the formats just defined because both the client and server have to have a code definition available to them of what each message will contain. But the system contains provisions for different protocol versions so that new servers can be brought into production still talking to other machines with an older protocol version until they can all be updated to the new one. They are very efficient, and pass binary data with no problem.

Compression

Since the time necessary to transmit data over the network is often more significant than the time your CPU spends preparing the data for transmission, it is often worthwhile to compress data before sending it. The popular HTTP protocol, as we will see in Chapter 9, lets a client and server figure out whether they can both support compression.

An interesting fact about the most ubiquitous form of compression, the GNU `zlib` facility that is available through the Python Standard Library, is that it is self-framing. If you start feeding it a compressed stream of data, then it can tell you when the compressed data has ended and further, uncompressed data has arrived past its end.

Most protocols choose to do their own framing and then, if desired, pass the resulting block to `zlib` for decompression. But you could conceivably promise yourself that you would always tack a bit of uncompressed data onto the end of each `zlib` compressed string—here, we will use a single `'.'` byte—and watch for your compression object to split out that "extra data" as the signal that you are done.

Consider this combination of two compressed data streams:

```
>>> import zlib
>>> data = zlib.compress('sparse') + '.' + zlib.compress('flat') + '.'
>>> data
'x\x9c+.H,*N\x05\x00\t\r\x02\x8f.x\x9cK\xcbI,\x01\x00\x04\x16\x01\xa8.'
>>> len(data)
28
```

Yes, I know, using 28 bytes to represent 10 actual characters of data is not terribly efficient; but this is just an example, and `zlib` works well only when given several dozen or more bytes of data to compress!

Imagine that these 28 bytes arrive at their destination in 8-byte packets. After processing the first packet, we will find the decompression object's `unused_data` slot still empty, which tells us that there is still more data coming, so we would `recv()` on our socket again:

```
>>> dobj = zlib.decompressobj()
>>> dobj.decompress(data[0:8]), dobj.unused_data
('spars', '')
```

But the second block of eight characters, when fed to our decompress object, both finishes out the compressed data we were waiting for (since the final `'e'` completes the string `'sparse'`) and also finally has a non-empty `unused_data` value that shows us that we finally received our `'.'` byte:

```
>>> dobj.decompress(data[8:16]), dobj.unused_data
('e', '.x')
```

If another stream of compressed data is coming, then we have to provide everything past the `'.'`—in this case, the character `'x'`—to our new decompress object, then start feeding it the remaining "packets":

```
>>> dobj2 = zlib.decompressobj()
>>> dobj2.decompress('x'), dobj2.unused_data
('', '')
>>> dobj2.decompress(data[16:24]), dobj2.unused_data
('flat', '')
>>> dobj2.decompress(data[24:]), dobj2.unused_data
('', '.')
```

At this point, `unused_data` is again non-empty, meaning that we have read past the end of this second bout of compressed data and can examine its content.

Again, most protocol designers make compression optional and simply do their own framing. But if you know ahead of time that you will always want to use `zlib`, then a convention like this would let you take advantage of the stream termination built into `zlib` and always detect the end of a compressed stream.

Network Exceptions

The example scripts in this book are generally designed to catch only those exceptions that are integral to the feature being demonstrated. So when we illustrated socket timeouts in Listing 2-2, we were careful to catch the exception `socket.timeout` since that is how timeouts are signaled; but we ignored all of the other exceptions that will occur if the hostname provided on the command line is invalid, or a remote IP is used with `bind()`, or the port used with `bind()` is already busy, or the peer cannot be contacted or stops responding.

What errors can result from working with sockets? Though the number of errors that can take place while using a network connection is quite large—involving every possible misstep that can occur at every stage of the complex TCP/IP protocol, for example—the number of actual exceptions with which socket operations can hit your programs is fortunately quite few. The exceptions that are specific to socket operations are:

- `socket.gaierror`: This exception is raised when `getaddrinfo()` cannot find a name or service that you ask about—hence the letters *G*, *A*, and *I* in its name! It can be raised not only when you make an explicit call to `getaddrinfo()`, but if you supply a hostname instead of an IP address to a call like `bind()` or `connect()` and the hostname lookup fails:

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect(('nonexistent.hostname.foo.bar', 80))
Traceback (most recent call last):
  ...
gaierror: [Errno -5] No address associated with hostname
```

- `socket.error`: This is the workhorse of the `socket` module, and will be raised for nearly every failure that can happen at any stage in a network transmission. Starting with Python 2.6, this exception became, appropriately enough, a subclass of the more general `IOError`. This can occur during nearly any socket call, even when you least expect it—because a previous `send()`, for example, might have elicited a reset (RST) packet from the remote host and the error will then be delivered whenever you next try to manipulate the socket.

- `socket.timeout`: This exception is raised only if you, or a library that you are using, decides to set a timeout on a socket rather than wait forever for a `send()` or `recv()` to complete. It indicates that the timeout was reached before the operation could complete normally.

You will see that the Standard Library documentation for the `socket` module also describes an `herror` exception; fortunately, it can occur only if you use certain old-fashioned address lookup calls instead of following the practices we outlined in Chapter 4.

A big question when you are using higher-level socket-based protocols from Python is whether they allow raw `socket` errors to hit your own code, or whether they catch them and turn them into their own kind of error.

Examples of both approaches exist within the Python Standard Library itself! For example, the `httplib` considers itself low-level enough that it can let you see the raw socket error that results from connecting to an unknown hostname:

```
>>> import httplib
>>> h = httplib.HTTPConnection('nonexistent.hostname.foo.bar')
>>> h.request('GET', '/')
Traceback (most recent call last):
  ...
gaierror: [Errno -2] Name or service not known
```

But the `urllib2`, probably because it wants to preserve the semantics of being a clean and neutral system for resolving URLs to documents, hides the very same error and returns a `URLError` instead:

```
>>> import urllib2
>>> urllib2.urlopen('http://nonexistent.hostname.foo.bar/')
Traceback (most recent call last):
  ...
URLError: <urlopen error [Errno -2] Name or service not known>
```

So depending on the protocol implementation that you are using, you might have to deal only with exceptions specific to that protocol, or you might have to deal with both protocol-specific exceptions and with raw `socket` errors as well. Consult documentation carefully if you are in doubt about the approach taken by a particular library. For the major packages that we cover in the subsequent chapters of this book, I have tried to provide insets that list the possible exceptions to which each library can subject your code.

And, of course, you can always fire up the library in question, provide it with a non-existent hostname (or simply run it when disconnected from the network!), and see what kind of exception comes out.

Handling Exceptions

When writing a network program, how should you handle all of the errors that can occur?

Of course, this question is not really specific to networking; all sorts of Python programs have to handle exceptions, and the techniques that we discuss briefly in this chapter are applicable to many other kinds of programs.

There are four basic approaches.

The first is not to handle exceptions at all. If only you or only other Python programmers will be using your script, then they will probably not be fazed by seeing an exception. Though they waste screen space and can make the reader squint to actually find the error message buried down at the bottom of the traceback, they are useful if the only recourse is likely to be editing the code to improve it!

If you are writing a library of calls to be used by other programmers, then this first approach is usually preferable, since by letting the exception through you give the programmer using your API the chance to decide how to present errors to his or her users. It is almost never appropriate for a library of code to make its own decision to terminate the program and print out a human-readable error message. What, for example, if the program is not running from the console and a pop-up window or system log message should be used instead?

But if you are indeed writing a library, then there is a second approach to consider: wrapping the network errors in an exception of your own. This can be very valuable if your library is complex—perhaps it maintains connections to several other services—and if it will be difficult for a programmer to guess which of the network operations that you are attempting resulted in the raw `socket.error` that you have allowed to be dumped in his or her lap.

If you offer a netcopy() method that copies a file from one remote machine to another, for example, a socket.error does not help the caller know whether the error was with the connection to the source machine, or the destination machine, or was some other problem altogether! In this case, it would be much better to define your own exceptions, like SourceHostError and DestHostError, which have a tight semantic relationship to the purpose of the netcopy API call that raised them. You can always include the original socket error as an instance variable of your own exception instances in case some users of your API will want to investigate further:

```
try:
»   host = sock.bind(address)
except socket.error as e:
»   raise URLError(e)
```

A third approach to exceptions is to wrap a try…except clause around every single network call that you ever make, and print out a pithy error message in its place. While suitable for short programs, this can become very repetitive when long programs are involved, without necessarily providing that much more information for the user. When you wrap the hundredth network operation in your program with yet another try…except and error message, ask yourself whether you are really providing that much more information than if you had just caught them all with one big exception handler.

And the idea of having big exception handlers that cover lots of code is the fourth and—in my opinion—best approach. Step back from your code and identify big regions that do specific things, like "this whole routine connects to the license server"; "all the socket operations in this function are fetching a response from the database"; and "this is all cleanup and shutdown code." Then the outer parts of your program—the ones that collect input, command-line arguments, and configuration, and then set big operations in motion—can wrap those big operations with handlers like the following:

```
try:
»   deliver_updated_keyfiles(…)
except (socket.error, socket.gaierror) as e:
»   print >>sys.stderr, 'cannot deliver remote keyfiles: %s' % (e)
»   exit(1)
```

Or, better yet, have pieces of code like this raise an error of your own devising:

```
except:
»   FatalError('cannot send replies: %s' % (e))
```

Then, at the very top level of your program, catch all of the FatalError exceptions that you throw and print the error messages out there. That way, when the day comes that you want to add a command-line option that sends fatal errors to the system error logs instead of to the screen, you have to adjust only one piece of code!

There is one final reason that might dictate where you add an exception handler to your network program: you might want to intelligently re-try an operation that failed. In long-running programs, this is common. Imagine a utility that periodically sent out e-mails with its status; if suddenly it cannot send them successfully, then it probably does not want to shut down for what might be just a transient error. Instead, the e-mail thread might log the error, wait several minutes, and try again.

In such cases, you will add exception handlers around very specific sequences of network operations that you want to treat as having succeeded or failed as a single combined operation. "If anything goes wrong in here, then I will just give up, wait ten minutes, and then start all over again the attempt to send that e-mail." Here the structure and logic of the network operations that you are performing—and not user or programmer convenience—will guide where you deploy try…except clauses.

Summary

For machine information to be placed on the network, it has to be transformed, so that whatever private and idiosyncratic storage mechanism is used inside your machine gets replaced by a public and reproducible representation that can be read on other systems, by other programs, and perhaps even by other programming languages.

For text, the big question will be choosing an encoding, so that the symbols you want to transmit can be changed into bytes, since 8-bit octets are the common currency of an IP network. Binary data will require your attention to make sure that bytes are ordered in a way that is compatible between different machines; the Python `struct` module will help you with this. Finally, data structures and documents are sometimes best sent using something like JSON or XML that provides a common way to share structured data between machines.

When using TCP/IP streams, a big question you will face is about framing: how, in the long stream of data, will you tell where a particular message starts and ends? There are many possible techniques for accomplishing this, all of which must be handled with care since `recv()` might return only part of an incoming transmission with each call. Special delimiter characters or patterns, fixed-length messages, and chunked-encoding schemes are all possible ways to festoon blocks of data so that they can be distinguished.

Not only will Python pickles transform data structures into strings that you can send across the network, but also the `pickle` module can tell where a pickle ends; this lets you use pickles not only to encode data but also to frame the individual messages on a stream. The `zlib` compression mechanism, which is often used with HTTP, also can tell when a compressed segment comes to an end, which can also provide you with inexpensive framing.

Sockets can raise several kinds of exceptions, as can network protocols that your code uses. The choice of when to use `try…except` clauses will depend on your audience—are you writing a library for other developers, or a tool for end users?—and also on semantics: you can wrap a whole section of your program in a `try…except` if all of that code is doing one big thing from the point of view of the caller or end user.

Finally, you will want to separately wrap operations with a `try…except` that can logically be re-tried in case the error is transient and they might later succeed.

CHAPTER 6

TLS and SSL

The short story is this: before you send sensitive data across a network, you need proof of the identity of the machine that you think is on the other end of the socket, and while sending the data, you need it protected against the prying eyes of anyone controlling the gateways and network switches that see all of your packets. The solution to this problem is to use Transport Layer Security (TLS). Because earlier versions of TLS were called the Secure Sockets Layer (SSL), nearly all of the libraries that you will use to speak TLS actually still have SSL somewhere in the name.

Simple enough?

The actual libraries introduced in this chapter, and all of the program listings that we will discuss, are going be about TLS; that is really the only take-home coding lesson you will find here. But we will do the actual examples last, once you have enough context to understand what they can—and cannot—do for the security of your network programs.

Computer Security

The subject of computer security is vast, with its own pundits, blogs, magazines, conferences, academic journals, consultancies, and product lines. And the subject is deserving of this extensive treatment: it is important, it is often done badly, and we programmers are still—as a profession—a very long way from even beginning to consistently get it right.

The field also has its share of charlatans and con men. And, of course, there are legions who work to subvert security mechanisms. Script kiddies seek surreptitious control over thousands of Windows desktops to power their next denial-of-service attack against their middle school. Organized criminals want to steal millions of dollars (or pounds sterling, or yen, or rupees, depending on their jurisdiction) through intercepted credit card numbers and identity theft. Militaries and governments seek to protect their own information systems while gaining the ability to commandeer or disable those of their rivals.

How can we competently dare to write programs that will be on the same network as such significant threats?

It may be foolish for me to attempt to answer this question; I should perhaps, at this point, be sending you right into the vast literature on the subject of secure software, and not risking offering you a few pointers when volumes of wisdom are what you really need. But since some readers may read this book on a tight schedule, tasked with adding TLS to some hastily written network program just days before it ships, I will go ahead and offer a few short thoughts on creating half-decent software.

First, always have thorough tests. Use Ned Batchelder's `coverage` tool to measure how much of your code is being tested at all. Freely refactor your code until testing each module is not just possible, but downright convenient! When a module is difficult to test, that is an excellent signal that the module has too many hard-coded entanglements with other parts of the code, and that it will be difficult to predict how the system as a whole will behave. The ease with which code can be tested, in other words, is often directly related to how easy it is to draw a boundary around which other parts of the system the code can directly affect.

Second, write as little code as possible. Rely on well-written and thoroughly tested third-party code whenever you can, with a special emphasis on using tools that seem to be well tested and actively maintained. One reason for using common technologies over obscure tools that you think might be better is that the code with the larger community is more likely to have its weaknesses and vulnerabilities discovered and resolved. Keep everything upgraded and up-to-date when possible, from the operating system and your Python install to the particular distributions you are using off of PyPI. And, of course, isolate your projects from each other by giving each of them its own virtual environment using the `virtualenv` command discussed in Chapter 1.

Third, the fact that you are reading this book indicates that you have probably already adopted one of my most important recommendations: to use a high-level language like Python for application development. Whole classes of security problems disappear when your code can talk directly about dictionaries, Unicode strings, and iteration over complex data structures, instead of having to manipulate raw integers every time it wants to visit every item in a list. Repetition and verbosity not only waste your time and cut your productivity, but also directly increase your chance of making a mistake.

Fourth, as you strive for elegant and simple solutions, try to learn as much as possible about the problem domain if many people have tackled it before you. Read about cross-scripting attacks (see Chapter 9) if you are writing a web site; about SQL injection attacks if your application talks to a database; about the sordid history of privilege escalation attacks if your system will support users who have different permission levels; and about viruses and Trojan horses if you are writing an e-mail client.

Fifth and finally, since you will probably lack the time (not to mention the omniscience) to build your entire application out of perfect code, try to focus on the edges of your code where it interacts with data from the outside. Several minutes spent writing code to examine a web form variable, checking it every which way to make sure it really and truly looks like, say, a string of digits, can be worth hours of precaution further inside the program that will be necessary if the bad value can make it all the way to the database and have to be dealt with there.

It was a great day, to take a concrete example, when C programmers stopped thinking that their servers had to always run as root—which had risked the compromise of the entire machine if something were to go wrong—and instead wrote network daemons that would start up, grab the low-numbered port on which their service lived, and then immediately drop their privileges to those of a normal user. They almost seemed to consider it a contest to see how few lines of code they could leave in the part of their program that ran with root privileges. And this brought about a vast reduction in exposure. Your Python code is the same way: fewer lines of code that run before you have verified the sanity of an input value, or tested for a subtle error, mean that less of the surface area of your program can harbor bugs that enemies could exploit.

But, again, the subject is a large one. Read blogs like "Schneier on Security," watch vendor security announcements like those on the Google Online Security Blog, and consult good books on the subject if you are going to be writing lots of security-sensitive code.

You should at least read lots of war stories about how intrusions have occurred, whether from security alerts or on popular blogs; then you will know ahead of time to forearm your code against the same kinds of disasters. Plus, such accounts are also quite entertaining if you like to learn the details of how systems work—and to learn about the unending cleverness of those who want to subvert them.

IP Access Rules

During the 1980s, the Internet grew from a small research network to a large enough community that it was unwise to trust everyone who happened to have access to an IP address. Prudence began to dictate that many services on each host either be turned off, or restricted so that only hosts in a pre-approved list were allowed to connect. But each piece of software had its own rules for how you specified the hosts that should be allowed to connect and the hosts whose connections should be rejected.

In 1990, Wietse Venema introduced the TCP Wrappers, and suggested that all Internet server programs could use this single piece of code to make access decisions. The idea was that rather than requiring every piece of software to have a separate configuration file, which made it impossible for

systems administrators to look any one place to discover exactly which remote services a particular machine was offering, a single pair of `hosts.allow` and `hosts.deny` files could be shared by many network services if each service looked for its own name (or the wildcard `ALL`).

It was soon discovered that rules were very difficult to maintain if they mixed arbitrary allow rules with specific deny rules naming hosts or IP address ranges that were thought to be dangerous—it meant staring at both `hosts.allow` and `hosts.deny` at the same time and trying to puzzle out the implications of both files for every possible IP address. So it quickly became popular to include only a single rule in `hosts.deny` that would disallow any connections that had not been explicitly permitted in the `hosts.allow` file:

```
ALL: ALL
```

The systems administrator could then focus on `hosts.allow`, safe in the knowledge that any hosts not explicitly mentioned there would be denied access. A typical `hosts.allow` looked something like this:

```
ALL: 127.0.0.1
portmap: 192.168.7
popd: 192.168
sshd: ALL
```

The ability to write rules like this was incredible. The `portmap` daemon in particular had long been a source of trouble. It was a necessary service if you were running the Network File System (NFS) to share files and directories between servers. But `portmap` had a long history of security problems, and it was very annoying to have to expose this service to everyone on the entire Internet just because a few nearby machines needed file sharing. Thanks to the TCP Wrappers, it was easy to lock down "dumb" network services like `portmap` that could not otherwise be configured to restrict the set of hosts that could connect.

If you remember those days, you might wonder what happened, and why a clean and uniform host filtering mechanism does not come built into Python.

There are several small reasons that contribute to this situation—most Python programs are not Internet daemons, for instance, so there has not been much pressure for such a mechanism in the Standard Library; and in a high-level language like Python, it is easy enough to pattern-match on IP addresses or hostnames that the burden of re-inventing this particular wheel for each project that needs it is not particularly high.

But I think there are two much bigger reasons.

First, many systems administrators these days simply use firewalls to limit remote host access instead of learning how to configure each and every daemon on their system (and then trusting that every one of those programs is going to actually implement their rules correctly). By putting basic access rules in the switches and gateways that form the fabric of an institution's network, and then implementing even more specific rules in each host's firewalls, system administrators get to configure a uniform and central set of controls upon network access.

But even more important is the fact that IP address restrictions are simply not effective as an ultimate security measure. If you want to control who has access to a resource, you need a stronger assurance of their identity these days than a simple check of the IP address from which their packets seem to originate.

While it is true that denial-of-service attacks still provide a good reason to have some basic IP-level access control rules enforced on your network—after all, if a service is needed only by other machines in the same server closet, why let everyone else even try to connect?—the proper place for such rules is, again, either the border firewall to an entire subnet, or the operating system firewall of each particular host. You really do not want your Python application code having to spin up for every single incoming connection from a denial-of-service attack, only to check the connection against a list of rules and then summarily reject it! Performing that check in the operating system, or on a network switch, is vastly more efficient.

If you do ever want to exercise some application-level IP access control in a particular program, simply examine the IP address returned by the accept() method on the socket with which your application is listening:

```
sc, sockname = s.accept()
if not sockname[0].startswith('192.168.'):
»   raise RuntimeError('connectors are not allowed from another network')
```

If you are interested in imposing the very specific restriction that only machines on your local subnet can connect to a particular service, but not machines whose packets are brought in through gateways, you might consider the SO_DONTROUTE option described in Chapter 2. But this restriction, like all rules based only on IP address, implies a very strong trust of the network hardware surrounding your machine—and therefore falls far short of the kind of assurance provided by TLS.

Finally, I note that the Ubuntu folks—who use Python in a number of their system and desktop services—maintain their own package for accessing libwrap0, a shared-library version of Wietse's old code, based on a Python package that was released on SourceForge in 2004. It allows them to do things like the following:

```
>>> from pytcpwrap.tcpwrap import TCPWrap
>>> TCPWrap('foo', None, '130.207.244.244').Allow()
False
```

But since this routine can be rather slow (it always does a reverse DNS lookup on the IP address), the Python code uses tabs and old-fashioned classes, and it has never been released on PyPI, I recommend against its use.

Cleartext on the Network

There are several security problems that TLS is designed to solve. They are best understood by considering the dangers of sending your network data as "cleartext" over a plain old socket, which copies your data byte-for-byte into the packets that get sent over the network.

Imagine that you run a typical web service consisting of front-end machines that serve HTML to customers and a back-end database that powers your service, and that all communication over your network is cleartext. What attacks are possible?

First, consider an adversary who can observe your packets as they travel across the network. This activity is called "network sniffing," and is quite legitimate when performed by network administrators trying to fix problems on their own hardware. The traditional program tcpdump and the more sleek and modern wireshark are both good tools if you want to try observing some network packets yourself.

Perhaps the adversary is sitting in a coffee shop, and he has a wireless card that is collecting your traffic as you debug one of the servers, and he keeps it for later analysis. Or maybe he has offered a bribe to a machine-room operator (or has gotten himself hired as a new operator!) and has attached a passive monitor to one of your network cables where it passes under the floor. But through whatever means, he can now observe, capture, and analyze your data at his leisure. What are the consequences?

- Obviously, he can see all of the data that passes over that segment of the network. The fraction of your data that he can capture depends on how much of it passes over that particular link. If he is watching conversations between your web front end and the database behind it, and only 1% of your customers log in every day to check their balances, then it will take him weeks to reconstruct a large fraction of your entire database. If, on the other hand, he can see the network segment that carries each night's disk backup to your mass storage unit, then in just a few hours he will learn the entire contents of your database.

- He will see any usernames and passwords that your clients use to connect to the servers behind them. Again, depending on which link he is observing, this might expose the passwords of customers signing on to use your service, or it might expose the passwords that your front ends use to get access to the database.
- Log messages can also be intercepted, if they are being sent to a central location and happen to travel over a compromised IP segment or device. This could be very useful if the observer wants to probe for vulnerabilities in your software: he can send illegal data to your server and watch for log messages indicating that he is causing errors; he will be forewarned about which activities of his are getting logged for your attention, and which you have neglected to log and that he can repeat as often as he wants; and, if your logs include tracebacks to help developers, then he will actually be able to view snippets of the code that he has discovered how to break to help him turn a bug into an actual compromise.
- If your database server is not picky about who connects, aside from caring that the web front end sends a password, then the attacker can now launch a “replay attack,” in which he makes his own connection to your database and downloads all of the data that a front-end server is normally allowed to access. If write permission is also granted, then rows can be adjusted, whole tables can be rewritten, or much of the database simply deleted, depending on the attacker's intentions.

Now we will take things to a second level: imagine an attacker who cannot yet alter traffic on your network itself, but who can compromise one of the services around the edges that help your servers find each other. Specifically, what if she can compromise the DNS service that lets your web front ends find your `db.example.com` server—or what if she can masquerade as your DNS server through a compromise at your upstream ISP? Then some interesting tricks might become possible:

- When your front ends ask for the hostname `db.example.com`, she could answer with the IP address of her own server, located anywhere in the world, instead. If the attacker has programmed her fake server to answer enough like your own database would, then she could collect at least the first batch of data—like a login name and maybe even a password—that arrives from each customer using your service.
- Of course, the fake database server will be at a loss to answer requests with any real data that the intruder has not already copied down off the network. Perhaps, if usernames and passwords are all she wanted, the attacker can just have the database not answer, and let your front-end service time out and then return an error to the user. True, this means that you will notice the problem; but if the attack lasts only about a minute or so and then your service starts working again, then you will probably blame the problem on a transient glitch and not suspect malfeasance. Meanwhile, the intruder may have captured dozens of user credentials.
- But if your database is not carefully locked down and so is not picky about which servers connect, then the attacker can do something more interesting: as requests start arriving at her fake database server, she can have it turn around and forward those requests to the *real* database server. This is called a “man-in-the-middle” attack. When the real answers come back, she can simply turn around and hand them back to the front-end services. Thus, without having actually compromised either your front-end web servers or the database server behind them, she will be in fairly complete control of your application: able to authenticate to the database because of the data coming in from the clients, and able to give convincing

answers back, thanks to her ability to connect to your database. Unlike the replay attack outlined earlier, this succeeds even if the clients are supplying a one-time password or are using a simple (though not a sophisticated) form of challenge-response.

- While proxying the client requests through to the database, the attacker will probably also have the option of inserting queries of her own into the request stream. This could let her download entire tables of data and delete or change whatever data the front-end services are typically allowed to modify.

Again, the man-in-the-middle attack is important because it can sometimes succeed without the need to actually compromise any of the servers involved, or even the network with which they are communicating—the attacker needs only to interfere with the naming service by which the servers discover each other.

Finally, consider an attacker who has actually compromised a router or gateway that stands between the various servers that are communicating in order to run your service. He will now be able to perform all of the actions that we just described—replay attacks, man-in-the-middle attacks, and all of the variations that allow him to insert or alter the database requests as they pass through the attacker's control—but will be able to do so without compromising the name service, or any of your services, and even if your database server is locked down to accept only connections from the real IP addresses of your front-end servers.

All of these evils are made possible by the fact that the clients and servers have no real guarantee, other than the IP addresses written openly into each packet, that they are really talking to each other.

TLS Encrypts Your Conversations

The secret to TLS is *public-key cryptography*, one of the great computing advances of the last few decades, and one of the very few areas of innovation in which academic computer science really shows its worth. There are several mathematical schemes that have been proved able to support public-key schemes, but they all have these three features:

- Anyone can generate a *key pair*, consisting of a private key that they keep to themselves and a public key that they can broadcast however they want. The public key can be shown to anyone in the world, because possessing the public key does not make it possible to derive or guess the private key. (Each key usually takes the physical form of a few kilobytes of binary data, often dumped into a text file using base64 or some other simple encoding.)

- If the public key is used to encrypt information, then the resulting block of binary data cannot be read by anyone, anywhere in the world, except by someone who holds the private key. This means that you can encrypt data with a public key and send it over a network with the assurance that no one but the holder of the corresponding private key will be able to read it.

- If the system that holds the private key uses it to encrypt information, then any copy of the public key can be used to decrypt the data. This does not make the data at all secret, of course, because we presume that anyone can get a copy of the public key; but it does prove that the information comes from the unique holder of the private key, since no one else could have generated data that the public key unlocks.

Following their invention, there have been many important applications developed for public-key cryptographic systems. I recommend Bruce Schneier's classic *Applied Cryptography* for a good

discussion of all of the ways that public keys can be used to help secure key-cards, protect individual documents, assert the identity of an e-mail author, and encrypt hard drives. Here, we will focus on how public keys are used in the TLS system.

Public keys are used at two different levels within TLS: first, to establish a certificate authority (CA) system that lets servers prove "who they really are" to the clients that want to connect; and, second, to help a particular client and server communicate securely. We will start by describing the lower level—how communication actually takes place—and then step back and look at how CAs work.

First, how can communication be protected against prying eyes in the first place?

It turns out that public-key encryption is pretty slow, so TLS does not actually use public keys to encrypt all of the data that you send over the network. Traditional symmetric-key encryption, where both sides share a big random block of data with which they encrypt outgoing traffic and decrypt incoming traffic, is much faster and better at handling large payloads. So TLS uses public-key cryptography only to begin each conversation: the server sends a public key to the client, the client sends back a suitable symmetric key by encrypting it with the public key, and now both sides hold the same symmetric key without an observer ever having been given the chance to capture it—since the observer will not be able to derive (thanks to powerful mathematics!) the server's private key based on seeing the public key go one way and an encrypted block of data going the other.

The actual TLS protocol involves a few other details, like the two partners figuring out the strongest symmetric key cipher that they both support (since new ones do get invented and added to the standard), but the previous paragraph gives you the gist of the operation.

And, by the way, the labels "server" and "client" here are rather arbitrary with respect to the actual protocol that you wind up speaking inside your encrypted socket—TLS has no way to actually know how you use the connection, or which side is the one that will be asking questions and which side will be answering. The terms "server" and "client" in TLS just mean that one end agrees to speak first and the other end will speak second when setting up the encrypted connection. There is only one important asymmetry built into the idea of a client and server, which we will learn about in a moment when we start discussing how the CA works.

So that is how your information is protected: a secret symmetric encryption key is exchanged using a public-private key pair, which is then used to protect your data in both directions. That alone protects your traffic against sniffing, since an attacker cannot see any of your data by watching from outside, and it also means that he cannot insert, delete, or alter the packets passing across a network node since, without the symmetric key, any change he makes to the data will simply produce gibberish when decrypted.

TLS Verifies Identities

But what about the other class of attacks we discussed—where an attacker gets you to connect to his server, and then talks to the real server to get the answers that you are expecting? That possibility is protected against by having a certificate authority, which we will now discuss.

Do you remember that the server end of a TLS connection starts by sharing a public key with the client? Well, it turns out that servers do not usually offer just any old public key—instead, they offer a public key that has been signed by a CA. To start up a certificate authority (some popular ones you might have heard of are Verisign, GeoTrust, and Thawte), someone simply creates a public-private key pair, publishes their public key far and wide, and then starts using their private key to "sign" server public keys by encrypting a hash of their data.

You will recall that only the holder of a private key can encrypt data that can then be decrypted with the corresponding public key; anyone else in the world who tries will wind up writing data that just turns into gibberish when passed through the public key. So when the client setting up a TLS connection receives a public key from the server along with a block of encrypted data that, when decrypted with the CA's public key, turns into a message that says "Go ahead and trust the server calling itself `db.example.com` whose public key hashes to the value `8A:01:1F:…`", then the client can trust that it is really connecting to `db.example.com` and not to some other server.

Thus man-in-the-middle attacks are thwarted, and it does not matter what tricks an attacker might use to rewrite packets or try to get you to connect to his server instead of the one that you really want to talk to. If he does not return to you the server's real certificate, then it will not really have been signed by the CA and your TLS library will tell you he is a fake; or, if the attacker *does* return the server's certificate—since, after all, it is publicly transmitted on the network—then your client will indeed be willing to start talking. But the first thing that your TLS library sends back will be the encrypted symmetric key that will govern the rest of the conversation—a key, alas, that the attacker cannot decrypt, because he does not possess the private key that goes along with the public server certificate that he is fraudulently waving around.

And, no, the little message that forms the digital signature does not really begin with the words "Go ahead" followed by the name of the server; instead, the server starts by creating a "certificate" that includes things like its name, an expiration date, and its public key, and the whole thing gets signed by the CA in a single step.

But how do clients learn about CA certificates? The answer is: configuration. Either you have to manually load them one by one (they tend to live in files that end in `.crt`) using a call to your SSL library, or perhaps the library you are using will come with some built in or that are provided by your operating system. Web browsers support HTTPS by coming with several dozen CA certificates, one for each major public CA in existence. These companies stake their reputations on keeping their private keys absolutely safe, and signing server certificates only after making absolutely sure that the request really comes from the owner of a given domain.

If you are setting up TLS servers that will be contacted only by clients that you configure, then you can save money by bypassing the public CAs and generating your own CA public-private key pair. Simply sign all of your server's certificates, and then put your new CA's public key in the configurations of all of your clients.

Some people go one step cheaper, and give their server a "self-signed" certificate that only proves that the public key being offered to the client indeed corresponds to a working private key. But a client that is willing to accept a self-signed certificate is throwing away one of the most important guarantees of TLS—that you are not talking to the wrong server—and so I strongly recommend that you set up your own simple CA in every case where spending money on "real" certificates from a public certificate authority does not make sense.

Guides to creating your own certificate authority can be found through your favorite search engine on the Web, as can software that automates the process so that you do not have to run all of those `openssl` command lines yourself.

Supporting TLS in Python

So how can you use TLS in your own code?

From the point of view of your network program, you start a TLS connection by turning control of a socket over to an SSL library. By doing so, you indicate that you want to stop using the socket for cleartext communication, and start using it for encrypted data under the control of the library.

From that point on, you no longer use the raw socket; doing so will cause an error and break the connection. Instead, you will use routines provided by the library to perform all communication. Both client and server should turn their sockets over to SSL at the same time, after reading all pending data off of the socket in both directions.

There are two general approaches to using SSL.

The most straightforward option is probably to use the `ssl` package that recent versions of Python ship with the Standard Library.

- The `ssl` package that comes with Python 3.2 includes everything that you need to communicate securely.

- The `ssl` packages that came with Python 2.6 through 3.1 neglected to provide a routine for actually verifying that server certificates match their hostname! For these Python versions, also install the `backports.ssl_match_hostname` distribution from the Python Package Index.
- For Python 2.5 and earlier, you will want to download both the `ssl` and `backports.ssl_match_hostname` distributions from the Python Package Index in order to have a complete solution.

The other alternative is to use a third-party Python library. There are several of these that support TLS, but many of them are decrepit and seem to have been abandoned.

The M2Crypto package is a happy exception. Although some people find it difficult to compile and install, it usually stays ahead of the Standard Library in letting you configure and control the security of your SSL connections. My own code examples that follow will use the Standard Library approach since I suspect that it will work for more people, but if you want more details the M2Crypto project is here:

`http://chandlerproject.org/bin/view/Projects/MeTooCrypto`

The project's author also has an interesting blog; you can see his posts about SSL in Python here:

`www.heikkitoivonen.net/blog/tag/ssl/`

Finally, you will want to avoid the Standard Library SSL support from Python 2.5. The `socket.ssl()` call that it supported—which was wisely removed before Python 2.6—provided no means of validating server certificates, and was therefore rather pointless. And its API was very awkward: the SSL object had a `read()` and `write()` method, but their semantics were those of `send()` and `recv()` on sockets, where it was possible for not all data to be sent, and you had to check the return value and possibly try again. I strongly recommend against its use.

The Standard SSL Module

Again, this module comes complete with Python 3.2, but it is missing a crucial function in earlier Python versions. For the Python versions covered by this book—versions 2.5 through 2.7—you will want to create a virtual environment (see Chapter 1) and run the following:

```
$ pip install backports.ssl_match_hostname
```

If you are using Python 2.5, then the `ssl` package itself also needs to be installed since that version of the Standard Library did not yet include it:

```
$ pip-2.5 install ssl
```

And, yes, in case you are curious, the "Brandon" who released that package is me—the very same one who has revised this book! For all of the other material in this volume, I was satisfied to merely report on the existing situation and try to point you toward the right solutions. But the SSL library situation was enough of a mess—with a simple enough solution—that I felt compelled to step in with the backport of the `match_hostname()` function before I could finish this chapter and be happy with the situation that it had to report.

Once you have those two tools, you are ready to use TLS! The procedure is simple and is shown in Listing 6–1. The first and last few lines of this file look completely normal: opening a socket to a remote server, and then sending and receiving data per the protocol that the server supports. The cryptographic protection is invoked by the few lines of code in the middle—two lines that load a certificate database and make the TLS connection itself, and then the call to `match_hostname()` that performs the crucial test of whether we are really talking to the intended server or perhaps to an impersonator.

Listing 6–1. *Wrapping a Client Socket with TLS Protection*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 6 - sslclient.py
# Using SSL to protect a socket in Python 2.6 or later

import os, socket, ssl, sys
from backports.ssl_match_hostname import match_hostname, CertificateError

try:
»   script_name, hostname = sys.argv
except ValueError:
»   print >>sys.stderr, 'usage: sslclient.py <hostname>'
»   sys.exit(2)

# First we connect, as usual, with a socket.

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, 443))

# Next, we turn the socket over to the SSL library!

ca_certs_path = os.path.join(os.path.dirname(script_name), 'certfiles.crt')
sslsock = ssl.wrap_socket(sock, ssl_version=ssl.PROTOCOL_SSLv3,
»   »   »   »   »   »     cert_reqs=ssl.CERT_REQUIRED, ca_certs=ca_certs_path)

# Does the certificate that the server proffered *really* match the
# hostname to which we are trying to connect?  We need to check.

try:
»   match_hostname(sslsock.getpeercert(), hostname)
except CertificateError, ce:
»   print 'Certificate error:', str(ce)
»   sys.exit(1)

# From here on, our `sslsock` works like a normal socket.  We can, for
# example, make an impromptu HTTP call.

sslsock.sendall('GET / HTTP/1.0\r\n\r\n')
result = sslsock.makefile().read()  # quick way to read until EOF
sslsock.close()
print 'The document https://%s/ is %d bytes long' % (hostname, len(result))
```

Note that the certificate database needs to be provided as a file named `certfiles.crt` in the same directory as the script; one such file is provided with the source code bundle that you can download for this book. I produced it very simply, by trusting the list of worldwide CAs that are trusted by default on my Ubuntu laptop, and combining these into a single file:

```
$ cat /etc/ssl/certs/* > certfiles.crt
```

Running Listing 6–1 against different web sites can demonstrate which ones provide correct certificates. For example, the OpenSSL web site does (as we would expect!):

```
$ python sslclient.py www.openssl.org
The document https://www.openssl.org/ is 15941 bytes long
```

The Linksys router here at my house, by contrast, uses a self-signed certificate that can provide encryption but fails to provide a signature that can be verified against any of the famous CAs in the `certfiles.crt` file. So, with the conservative settings in our `sslclient.py` program, the connection fails:

```
$ python sslclient.py ten22.rhodesmill.org
Traceback (most recent call last):
  ...
ssl.SSLError: [Errno 1] _ssl.c:480: error:14090086:SSL
routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
```

Interestingly, Google (as of this writing) provides a single `www.google.com` certificate not only for that specific domain name, but also for its `google.com` address since all that is hosted there is a redirect to the `www` name:

```
$ python sslclient.py google.com
Certificate error: hostname 'google.com' doesn't match u'www.google.com'
$ python sslclient.py www.google.com
The document https://www.google.com/ is 9014 bytes long
```

Writing an SSL server looks much the same: code like that in Listing 3-1 is supplemented so that the client socket returned by each `accept()` call gets immediately wrapped with `wrap_socket()`, but with different options than are used with a client. In general, here are the three most popular ways of using `wrap_socket()` (see the `ssl` Standard Library documentation to learn about all of the rest of its options):

The first form is the one shown in Listing 6–1, and is the most common form of the call seen in clients:

```
wrap_socket(sock, ssl_version=ssl.PROTOCOL_SSLv3,
»   cert_reqs=ssl.CERT_REQUIRED, ca_certs=ca_certs_path)
```

Here the client asserts no particular identity—at least, TLS provides no way for the server to know who is connecting. (Since the connection is now encrypted, of course, a password or cookie can now be passed safely to the server; but the TLS layer itself will not know who the client is.)

Servers generally do not care whether clients connect with certificates, so the `wrap_socket()` calls that they make after an `accept()` use a different set of named parameters that provide the documents that establish their own identity. But they can neglect to provide a database of CA certificates, since they will not require the client to present a certificate:

```
wrap_socket(sock, server_side=True, ssl_version=ssl.PROTOCOL_SSLv23,
»   cert_reqs=ssl.CERT_NONE,
»   keyfile="mykeyfile", certfile="mycertfile")
```

Finally, there do exist situations where you want to run a server that checks the certificates of the clients that are connecting. This can be useful if the protocol that you are wrapping provides weak or even non-existent authentication, and the TLS layer will be providing the only assurance about who is connecting. You will use your CA to sign client certificates for each individual or machine that will be connecting, then have your server make a call like this:

```
wrap_socket(sock, server_side=True, ssl_version=ssl.PROTOCOL_SSLv23,
»   cert_reqs=ssl.CERT_REQUIRED, ca_certs=ca_certs_path,
»   keyfile="mykeyfile", certfile="mycertfile")
```

Again, consult the `ssl` chapter in the Standard Library if you need to delve into the options more deeply; the documentation there has been getting quite a bit better, and might cover edge cases that we have not had room to discuss here in this chapter.

If you are writing clients and servers that need to talk only to each other, try using `PROTOCOL_TLSv1` as your protocol. It is more modern and secure than any of the protocols that have `SSL` in their names. The only reason to use `SSL` protocols—as shown in the foregoing example calls, and which are also currently

the defaults for the `wrap_socket()` call in the Standard Library—is if you need to speak to browsers or other third-party clients that might not have upgraded to full-fledged TLS yet.

Loose Ends

When adding cryptography to your application, it is always a good idea to read up-to-date documentation. The advice given in this chapter would have been quite different if this revision of the book had happened even just one or two years earlier, and in two or three more years it will doubtless be out of date.

In particular, the idea has been around for a long time in the public-key cryptography literature that there should exist certificate *revocation lists*, where client certificates and even certificate-authority certificates could be listed if they are discovered to have been compromised and must no longer be trusted. That way, instead of everyone waiting for operating system updates or browser upgrades to bring the news that an old CA certificate should no longer be trusted, they could instantly be protected against any client certificates minted with the stolen private key.

Also, security vulnerabilities continue to be discovered not only in particular programs but also in the design of various security protocols themselves—SSL version 2 was, in fact, the victim of just such a discovery in the mid-1990s, which is why many people simply turn it off as an option when using TLS.

All of which is to say: use this chapter as a basic API reference and introduction to the whole topic of secure sockets, but consult something more up-to-date if you are creating new software more than a year after this book comes out, to make sure the protocols still operate well if used as shown here. As of this writing, the Standard Library documentation, Python blogs, and Stack Overflow questions about cryptography are all good places to look.

Summary

Computer security is a large and complicated subject. At its core is the fact that an intruder or troublemaker will take advantage of almost any mistake you make—even an apparently very small one—to try to leverage control over your systems and software.

Networks are the locus of much security effort because the IP protocols, by default, copy all your information into packets verbatim, where it can be read by anyone watching your packets go past. Passive sniffing, man-in-the-middle attacks, connection hijacking, and replay attacks are all possible if an adversary has control over the network between a client and server.

Fortunately, mathematicians have invented public-key cryptography, which has been packaged as the TLS protocol for protecting IP sockets. It grew out of an older, less secure protocol named SSL, from which most software libraries that speak TLS take their name.

The Python Standard Library now supplies an `ssl` package (though it has to be downloaded separately for Python 2.5), which can leverage the OpenSSL library to secure your own application sockets. This makes it impossible for a third party to masquerade as a properly certified server machine, and also encrypts all data so that an observer cannot determine what your client and server programs are saying to one another.

There are two keys to using the `ssl` package. First, you should always wrap the bare socket you create with its `wrap_socket()` function, giving the right arguments for the kind of connection and certificate assurances that you need. Second, if you expect the other side to provide a certificate, then you should run `match_hostname()` to make sure that they are claiming the identity that you expect.

The security playing field shifts every few years, with old protocols obsoleted and new ones developed, so keep abreast of news if you are writing security-sensitive applications.

CHAPTER 7

Server Architecture

This chapter explores how network programming intersects with the general tools and techniques that Python developers use to write long-running daemons that can perform significant amounts of work by keeping a computer and its processors busy.

Instead of making you read through this entire chapter to learn the landscape of design options that I will explore, let me outline them quickly.

Most of the network programs in this book—and certainly all of the ones you have seen so far—use a single sequence of Python instructions to serve one network client at a time, operating in lockstep as requests come in and responses go out. This, as we will see, will usually leave the system CPU mostly idle.

There are two changes you can make to a network program to improve this situation, and then a third, big change that you can make outside your program that will allow it to scale even further.

The two changes you can make to your program are either to rewrite it in an event-driven style that can accept several client connections at once and then answer whichever one is ready for an answer next, or to run several copies of your single-client server in separate threads or processes. An event-driven style does not impose the expense of operating system context switches, but, on the other hand, it can saturate at most only one CPU, whereas multiple threads or processes—and, with Python, especially processes—can keep all of your CPU cores busy handling client requests.

But once you have crafted your server so that it keeps a single machine perfectly busy answering clients, the only direction in which you can expand is to balance the load of incoming connections across several different machines, or even across data centers. Some large Internet services do this with proxy devices sitting in front of their server racks; others use DNS round-robin, or nameservers that direct clients to servers in the same geographic location; and we will briefly discuss both approaches later in this chapter.

Daemons and Logging

Part of the task of writing a network daemon is, obviously, the part where you write the program as a daemon rather than as an interactive or command-line tool. Although this chapter will focus heavily on the "network" part of the task, a few words about general daemon programming seem to be in order.

First, you should realize that creating a daemon is a bit tricky and can involve a dozen or so lines of code to get completely correct. And that estimate assumes a POSIX operating system; under Windows, to judge from the code I have seen, it is even more difficult to write what is called a "Windows service" that has to be listed in the system registry before it can even run.

On POSIX systems, rather than cutting and pasting code from a web site, I encourage you to use a good Python library to make your server a daemon. The official purpose of becoming a daemon, by the way, is so that your server can run independently of the terminal window and user session that were used to launch it. One approach toward running a service as a daemon—the one, in fact, that I myself prefer—is to write a completely normal Python program and then use Chris McDonough's `supervisord` daemon to start and monitor your service. It can even do things like re-start your program if it should die, but then give up if several re-starts happen too quickly; it is a powerful tool, and worth a good long look: `http://supervisord.org/`.

You can also install python-daemon from the Package Index (a module named daemon will become part of the Standard Library in Python 3.2), and its code will let your server program become a daemon entirely on its own power.

If you are running under supervisord, then your standard output and error can be saved as rotated log files, but otherwise you will have to make some provision of your own for writing logs. The most important piece of advice that I can give in that case is to avoid the ancient syslog Python module, and use the modern logging module, which can write to syslog, files, network sockets, or anything in between. The simplest pattern is to place something like this at the top of each of your daemon's source files:

```
import logging
log = logging.getLogger(__name__)
```

Then your code can generate messages very simply:

```
log.error('the system is down')
```

This will, for example, induce a module that you have written that is named serv.inet to produce log messages under its own name, which users can filter either by writing a specific serv.inet handler, or a broader serv handler, or simply by writing a top-level rule for what happens to all log messages. And if you use the logger module method named fileConfig() to optionally read in a logging.conf provided by your users, then you can leave the choice up to them about which messages they want recorded where. Providing a file with reasonable defaults is a good way to get them started.

For information on how to get your network server program to start automatically when the system comes up and shut down cleanly when your computer halts, check your operating system documentation; on POSIX systems, start by reading the documentation surrounding your operating system's chosen implementation of the "init scripts" subsystem.

Our Example: Sir Launcelot

I have designed a very simple network service to illustrate this chapter so that the details of the actual protocol do not get in the way of explaining the server architectures. In this minimalist protocol, the client opens a socket, sends across one of the three questions asked of Sir Launcelot at the Bridge of Death in Monty Python's *Holy Grail* movie, and then terminates the message with a question mark:

```
What is your name?
```

The server replies by sending back the appropriate answer, which always ends with a period:

```
My name is Sir Launcelot of Camelot.
```

Both question and answer are encoded as ASCII.

Listing 7–1 defines two constants and two functions that will be very helpful in keeping our subsequent program listings short. It defines the port number we will be using; a list of question-answer pairs; a recv_until() function that keeps reading data from a network socket until it sees a particular piece of punctuation (or any character, really, but we will always use it with either the '.' or '?' character); and a setup() function that creates the server socket.

Listing 7–1. *Constants and Functions for the Launcelot Protocol*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - launcelot.py
# Constants and routines for supporting a certain network conversation.

import socket, sys
```

```
PORT = 1060
qa = (('What is your name?', 'My name is Sir Launcelot of Camelot.'),
»     ('What is your quest?', 'To seek the Holy Grail.'),
»     ('What is your favorite color?', 'Blue.'))
qadict = dict(qa)

def recv_until(sock, suffix):
»   message = ''
»   while not message.endswith(suffix):
»   »   data = sock.recv(4096)
»   »   if not data:
»   »   »   raise EOFError('socket closed before we saw %r' % suffix)
»   »   message += data
»   return message

def setup():
»   if len(sys.argv) != 2:
»   »   print >>sys.stderr, 'usage: %s interface' % sys.argv[0]
»   »   exit(2)
»   interface = sys.argv[1]
»   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
»   sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
»   sock.bind((interface, PORT))
»   sock.listen(128)
»   print 'Ready and listening at %r port %d' % (interface, PORT)
»   return sock
```

Note in particular that the `recv_until()` routine does not require its caller to make any special check of its return value to discover whether an end-of-file has occurred. Instead, it raises `EOFError` (which in Python itself is raised only by regular files) to indicate that no more data is available on the socket. This will make the rest of our code a bit easier to read.

With the help of these routines, and using the same TCP server pattern that we learned in Chapter 3, we can construct the simple server shown in Listing 7–2 using only a bit more than a dozen lines of code.

Listing 7–2. *Simple Launcelot Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - server_simple.py
# Simple server that only serves one client at a time; others have to wait.

import launcelot

def handle_client(client_sock):
»   try:
»   »   while True:
»   »   »   question = launcelot.recv_until(client_sock, '?')
»   »   »   answer = launcelot.qadict[question]
»   »   »   client_sock.sendall(answer)
»   except EOFError:
»   »   client_sock.close()

def server_loop(listen_sock):
»   while True:
```

```
»   »   client_sock, sockname = listen_sock.accept()
»   »   handle_client(client_sock)

if __name__ == '__main__':
»   listen_sock = launcelot.setup()
»   server_loop(listen_sock)
```

Note that the server is formed of two nested loops. The outer loop, conveniently defined in a function named `server_loop()` (which we will use later in some other program listings), forever accepts connections from new clients and then runs `handle_client()` on each new socket—which is itself a loop, endlessly answering questions that arrive over the socket, until the client finally closes the connection and causes our `recv_until()` routine to raise `EOFError`.

By the way, you will see that several listings in this chapter use additional ink and whitespace to include `__name__ == '__main__'` stanzas, despite my assertion in the preface that I would not normally do this in the published listings. The reason, as you will soon discover, is that some of the subsequent listings import these earlier ones to avoid having to repeat code. So the result, overall, will be a savings in paper!

Anyway, this simple server has terrible performance characteristics.

What is wrong with the simple server? The difficulty comes when many clients all want to connect at the same time. The first client's socket will be returned by `accept()`, and the server will enter the `handle_client()` loop to start answering that first client's questions. But while the questions and answers are trundling back and forth across the network, all of the other clients are forced to queue up on the queue of incoming connections that was created by the `listen()` call in the `setup()` routine of Listing 7–1.

The clients that are queued up cannot yet converse with the server; they remain idle, waiting for their connection to be accepted so that the data that they want to send can be received and processed.

And because the waiting connection queue itself is only of finite length—and although we asked for a limit of 128 pending connections, some versions of Windows will actually support a queue only 5 items long—if enough incoming connections are attempted while others are already waiting, then the additional connections will either be explicitly refused or, at least, quietly ignored by the operating system. This means that the three-way TCP handshakes with these additional clients (we learned about handshakes in Chapter 3) cannot even commence until the server has finished with the first client and accepted another waiting connection from the listen queue.

An Elementary Client

We will tackle the deficiencies of the simple server shown in Listing 7–2 in two discussions. First, in this section, we will discuss how much time it spends waiting even on one client that needs to ask several questions; and in the next section, we will look at how it behaves when confronted with many clients at once.

A simple client for the Launcelot protocol is shown in Listing 7–3. It connects, asks each of the three questions once, and then disconnects.

Listing 7–3. *A Simple Three-Question Client*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - client.py
# Simple Launcelot client that asks three questions then disconnects.

import socket, sys, launcelot

def client(hostname, port):
```

```
»   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
»   s.connect((hostname, port))
»   s.sendall(launcelot.qa[0][0])
»   answer1 = launcelot.recv_until(s, '.')  # answers end with '.'
»   s.sendall(launcelot.qa[1][0])
»   answer2 = launcelot.recv_until(s, '.')
»   s.sendall(launcelot.qa[2][0])
»   answer3 = launcelot.recv_until(s, '.')
»   s.close()
»   print answer1
»   print answer2
»   print answer3

if __name__ == '__main__':
»   if not 2 <= len(sys.argv) <= 3:
»   »   print >>sys.stderr, 'usage: client.py hostname [port]'
»   »   sys.exit(2)
»   port = int(sys.argv[2]) if len(sys.argv) > 2 else launcelot.PORT
»   client(sys.argv[1], port)
```

With these two scripts in place, we can start running our server in one console window:

```
$ python server_simple.py localhost
```

We can then run our client in another window, and see the three answers returned by the server:

```
$ python client.py localhost
My name is Sir Launcelot of Camelot.
To seek the Holy Grail.
Blue.
```

The client and server run very quickly here on my laptop. But appearances are deceiving, so we had better approach this client-server interaction more scientifically by bringing real measurements to bear upon its activity.

The Waiting Game

To dissect the behavior of this server and client, I need two things: more realistic network latency than is produced by making connections directly to `localhost`, and some way to see a microsecond-by-microsecond report on what the client and server are doing.

These two goals may initially seem impossible to reconcile. If I run the client and server on the same machine, the network latency will not be realistic. But if I run them on separate servers, then any timestamps that I print will not necessarily agree because of slight differences between the machines' clocks.

My solution is to run the client and server on a single machine (my Ubuntu laptop, in case you are curious) but to send the connection through a round-trip to another machine (my Ubuntu desktop) by way of an SSH tunnel. See Chapter 16 and the SSH documentation itself for more information about tunnels. The idea is that SSH will open local port 1061 here on my laptop and start accepting connections from clients. Each connection will then be forwarded across to the SSH server running on my desktop machine, which will connect back using a normal TCP connection to port 1060 here on my laptop, whose IP ends with `.5.130`. Setting up this tunnel requires one command, which I will leave running in a terminal window while this example progresses:

```
$ ssh -N -L 1061:192.168.5.130:1060 kenaniah
```

Now that I can build a connection between two processes on this laptop that will have realistic latency, I can build one other tool: a Python source code tracer that measures when statements run with microsecond accuracy. It would be nice to have simply been able to use Python's `trace` module from the Standard Library, but unfortunately it prints only hundredth-of-a-second timestamps when run with its `-g` option.

And so I have written Listing 7–4. You give this script the name of a Python function that interests you and the name of the Python program that you want to run (followed by any arguments that it takes); the tracing script then runs the program and prints out every statement inside the function of interest just before it executes. Each statement is printed along with the current second of the current minute, from zero to sixty. (I omitted minutes, hours, and days because such long periods of time are generally not very interesting when examining a quick protocol like this.)

Listing 7–4. *Tracer for a Python Function*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - my_trace.py
# Command-line tool for tracing a single function in a program.

import linecache, sys, time

def make_tracer(funcname):
    def mytrace(frame, event, arg):
        if frame.f_code.co_name == funcname:
            if event == 'line':
                _events.append((time.time(), frame.f_code.co_filename,
                                                frame.f_lineno))
            return mytrace
    return mytrace

if __name__ == '__main__':
    _events = []
    if len(sys.argv) < 3:
        print >>sys.stderr, 'usage: my_trace.py funcname other_script.py ...'
        sys.exit(2)
    sys.settrace(make_tracer(sys.argv[1]))
    del sys.argv[0:2]  # show the script only its own name and arguments
    try:
        execfile(sys.argv[0])
    finally:
        for t, filename, lineno in _events:
            s = linecache.getline(filename, lineno)
            sys.stdout.write('%9.6f  %s' % (t % 60.0, s))
```

Note that the tracing routine is very careful not to perform any expensive I/O as parts of its activity; it neither retrieves any source code, nor prints any messages while the subordinate script is actually running. Instead, it saves the timestamps and code information in a list. When the program finishes running, the `finally` clause runs leisurely through this data and produces output without slowing up the program under test.

We now have all of the pieces in place for our trial! We first start the server, this time inside the tracing program so that we will get a detailed log of how it spends its time inside the `handle_client()` routine:

```
$ python my_trace.py handle_client server_simple.py ''
```

Note again that I had it listen to the whole network with `''`, and not to any particular interface, because the connections will be arriving from the SSH server over on my desktop machine. Finally, I can run a traced version of the client that connects to the forwarded port 1061:

```
$ python my_trace.py client client.py localhost 1061
```

The client prints out its own trace as it finishes. Once the client finished running, I pressed Ctrl+C to kill the server and force it to print out its own trace messages. Both machines were connected to my wired network for this test, by the way, because its performance is much better than that of my wireless network.

Here is the result. I have eliminated a few extraneous lines—like the `try` and `while` statements in the server loop—to make the sequence of actual network operations clearer, and I have indented the server's output so that we can see how its activities interleaved with those of the client. Again, it is because they were running on the same machine that I can so confidently trust the timestamps to give me a strict ordering:

```
Client /
   Server  (times in seconds)
-------------------------------------------------------------------
14.225574      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14.225627      s.connect((hostname, port))
14.226107      s.sendall(launcelot.qa[0][0])
14.226143      answer1 = launcelot.recv_until(s, '.')  # answers end with '.'
   14.227495          question = launcelot.recv_until(client_sock, '?')
   14.228496          answer = launcelot.qadict[question]
   14.228505          client_sock.sendall(answer)
   14.228541          question = launcelot.recv_until(client_sock, '?')
14.229348      s.sendall(launcelot.qa[1][0])
14.229385      answer2 = launcelot.recv_until(s, '.')
   14.229889          answer = launcelot.qadict[question]
   14.229898          client_sock.sendall(answer)
   14.229929          question = launcelot.recv_until(client_sock, '?')
14.230572      s.sendall(launcelot.qa[2][0])
14.230604      answer3 = launcelot.recv_until(s, '.')
   14.231200          answer = launcelot.qadict[question]
   14.231207          client_sock.sendall(answer)
   14.231237          question = launcelot.recv_until(client_sock, '?')
14.231956      s.close()
   14.232651          client_sock.close()
```

When reading this trace, keep in mind that having tracing turned on will have made both programs slower; also remember that each line just shown represents the moment that Python arrived at each statement and *started* executing it. So the expensive statements are the ones with long gaps between their own timestamp and that of the *following* statement.

Given those caveats, there are several important lessons that we can learn from this trace.

First, it is plain that the very first steps in a protocol loop can be different than the pattern into which the client and server settle once the exchange has really gotten going. For example, you can see that Python reached the server's `question =` line twice during its first burst of activity, but only once per iteration thereafter. To understand the steady state of a network protocol, it is generally best to look at the very middle of a trace like this where the pattern has settled down and measure the time it takes the protocol to go through a cycle and wind up back at the same statement.

Second, note how the cost of communication dominates the performance. It always seems to take less than 10 μs for the server to run the `answer =` line and retrieve the response that corresponds to a particular question. If actually generating the answer were the client's only job, then we could expect it to serve more than 100,000 client requests per second!

But look at all of the time that the client and server spend waiting for the network: every time one of them finishes a `sendall()` call, it takes between 500 µs and 800 µs before the other conversation partner is released from its `recv()` call and can proceed. This is, in one sense, very little time; when you can ping another machine and get an answer in around 1.2 ms, you are on a pretty fast network. But the cost of the round-trip means that, if the server simply answers one question after another, then it can answer at most around 1,000 requests per second—only one-hundredth the rate at which it can generate the answers themselves!

So the client and server both spend most of their time waiting. And given the lockstep single-threaded technique that we have used to design them, they cannot use that time for anything else.

A third observation is that the operating system is really very aggressive in taking tasks upon itself and letting the programs go ahead and get on with their lives—a feature that we will use to great advantage when we tackle event-driven programming. Look, for example, at how each `sendall()` call uses only a few dozen microseconds to queue up the data for transmission, and then lets the program proceed to its next instruction. The operating system takes care of getting the data actually sent, without making the program wait.

Finally, note the wide gulfs of time that are involved in simply setting up and tearing down the socket. Nearly 1,900 µs pass between the client's initial `connect()` and the moment when the server learns that a connection has been accepted and that it should start up its `recv_until()` routine. There is a similar delay while the socket is closed down. This leads to designers adding protocol features like the keep-alive mechanism of the HTTP/1.1 protocol (Chapter 9), which, like our little Launcelot protocol here, lets a client make several requests over the same socket before it is closed.

So if we talk to only one client at a time and patiently wait on the network to send and receive each request, then we can expect our servers to run hundreds or thousands of times more slowly than if we gave them more to do. Recall that a modern processor can often execute more than 2,000 machine-level instructions per microsecond. That means that the 500 µs delay we discussed earlier leaves the server idle for nearly a half-million clock cycles before letting it continue work!

Through the rest of this chapter, we will look at better ways to construct servers in view of these limitations.

Running a Benchmark

Having used microsecond tracing to dissect a simple client and server, we are going to need a better system for comparing the subsequent server designs that we explore. Not only do we lack the space to print and analyze increasingly dense and convoluted timestamp traces, but that approach would make it very difficult to step back and to ask, "Which of these server designs is working the best?"

We are therefore going to turn now to a public tool: the FunkLoad tool, written in Python and available from the Python Package Index. You can install it in a virtual environment (see Chapter 1) with a simple command:

```
$ pip install funkload
```

There are other popular benchmark tools available on the Web, including the "Apache bench" program named `ab`, but for this book it seemed that the leading Python load tester would be a good choice.

FunkLoad can take a test routine and run more and more copies of it simultaneously to test how the resources it needs struggle with the rising load. Our test routine will be an expanded version of the simple client that we used earlier: it will ask ten questions of the server instead of three, so that the network conversation itself will take up more time relative to the TCP setup and teardown times that come at the beginning and end. Listing 7–5 shows our test routine, embedded in a standard `unittest` script that we can also run on its own.

Listing 7–5. *Test Routine Prepared for Use with FunkLoad*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - launcelot_tests.py
# Test suite that can be run against the Launcelot servers.

from funkload.FunkLoadTestCase import FunkLoadTestCase
import socket, os, unittest, launcelot

SERVER_HOST = os.environ.get('LAUNCELOT_SERVER', 'localhost')

class TestLauncelot(FunkLoadTestCase):
»   def test_dialog(self):
»   »   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
»   »   sock.connect((SERVER_HOST, launcelot.PORT))
»   »   for i in range(10):
»   »   »   question, answer = launcelot.qa[i % len(launcelot.qa)]
»   »   »   sock.sendall(question)
»   »   »   reply = launcelot.recv_until(sock, '.')
»   »   »   self.assertEqual(reply, answer)
»   »   sock.close()

if __name__ == '__main__':
»   unittest.main()
```

The IP address to which the test client connects defaults to `localhost` but can be adjusted by setting a `LAUNCELOT_SERVER` environment variable (since I cannot see any way to pass actual arguments through to tests with FunkLoad command-line arguments).

Because FunkLoad itself, like other load-testing tools, can consume noticeable CPU, it is always best to run it on another machine so that its own activity does not slow down the server under test. Here, I will use my laptop to run the various server programs that we consider, and will run FunkLoad over on the same desktop machine that I used earlier for building my SSH tunnel. This time there will be no tunnel involved; FunkLoad will hit the server directly over raw sockets, with no other pieces of software standing in the way.

So here on my laptop, I run the server, giving it a blank interface name so that it will accept connections on any network interface:

```
$ python server_simple.py ''
```

And on the other machine, I create a small FunkLoad configuration file, shown in Listing 7–6, that arranges a rather aggressive test with an increasing number of test users all trying to make repeated connections to the server at once—where a "user" simply runs, over and over again, the test case that you name on the command line. Read the FunkLoad documentation for an explanation, accompanied by nice ASCII-art diagrams, of what the various parameters mean.

Listing 7–6. *Example FunkLoad Configuration*

```
# TestLauncelot.conf
[main]
title=Load Test For Chapter 7
description=From the Foundations of Python Network Programming
url=http://localhost:1060/

[ftest]
log_path = ftest.log
```

```
result_path = ftest.xml
sleep_time_min = 0
sleep_time_max = 0

[bench]
log_to = file
log_path = bench.log
result_path = bench.xml
cycles = 1:2:3:5:7:10:13:16:20
duration = 8
startup_delay = 0.1
sleep_time = 0.01
cycle_time = 10
sleep_time_min = 0
sleep_time_max = 0
```

Note that FunkLoad finds the configuration file name by taking the class name of the test case—which in this case is `TestLauncelot`—and adding `.conf` to the end. If you re-name the test, or create more tests, then remember to create corresponding configuration files with those class names.

Once the test and configuration file are in place, the benchmark can be run. I will first set the environment variable that will alert the test suite to the fact that I want it connecting to another machine. Then, as a sanity check, I will run the test client once as a normal test to make sure that it succeeds:

```
$ export LAUNCELOT_SERVER=192.168.5.130
$ fl-run-test launcelot_tests.py TestLauncelot.test_dialog
.
----------------------------------------------------------------------
Ran 1 test in 0.228s
OK
```

You can see that FunkLoad simply expects us to specify the Python file containing the test, and then specify the test suite class name and the test method separated by a period. The same parameters are used when running a benchmark:

```
$ fl-run-bench launcelot_tests.py TestLauncelot.test_dialog
```

The result will be a `bench.xml` file full of XML (well, nobody's perfect) where FunkLoad stores the metrics generated during the test, and from which you can generate an attractive HTML report:

```
$ fl-build-report --html bench.xml
```

Had we been testing a web service, the report would contain several different analyses, since FunkLoad would be aware of how many web pages each iteration of the test had downloaded. But since we are not using any of the web-specific test methods that FunkLoad provides, it cannot see inside our code and determine that we are running ten separate requests inside every connection. Instead, it can simply count how many times each test runs per second; the result is shown in Figure 7–1.

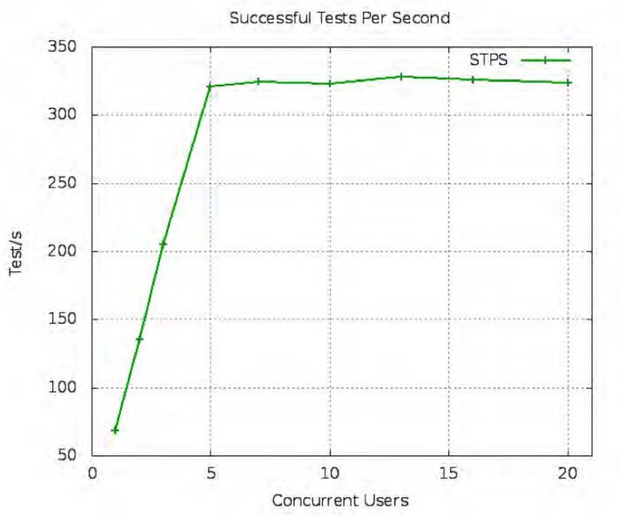


Figure 7–1. *The performance of our simple server*

Since we are sending ten Launcelot questions per test trial, the 325 test-per-second maximum that the simple server reaches represents 3,250 questions and answers—more than the 1,000 per second that we guessed were possible when testing `server_simple.py` over the slower SSH tunnel, but still of the same order of magnitude.

In interpreting this report, it is critical to understand that a healthy graph shows a linear relationship between the number of requests being made and the number of clients that are waiting. This server shows great performance all the way up to five clients. How can it be improving its performance, when it is but a single thread of control stuck talking to only one client at a time? The answer is that having several clients going at once lets one be served while another one is still tearing down its old socket, and yet another client is opening a fresh socket that the operating system will hand the server when it next calls `accept()`.

But the fact that sockets can be set up and torn down at the same time as the server is answering one client's questions only goes so far. Once there are more than five clients, disaster strikes: the graph flatlines, and the increasing load means that a mere 3,250 answers per second have to be spread out over 10 clients, then 20 clients, and so forth. Simple division tells us that 5 clients see 650 questions answered per second; 10 clients, 325 questions; and 20 clients, 162 questions per second. Performance is dropping like a rock.

So that is the essential limitation of this first server: when enough clients are going at once that the client and server operating systems can pipeline socket construction and socket teardown in parallel, the server's insistence on talking to only one client at a time becomes the insurmountable bottleneck and no further improvement is possible.

Event-Driven Servers

The simple server we have been examining has the problem that the `recv()` call often finds that no data is yet available from the client, so the call "blocks" until data arrives. The time spent waiting, as we have seen, is time lost; it cannot be spent usefully by the server to answer requests from other clients.

But what if we avoided ever calling recv() until we knew that data had arrived from a particular client—and, meanwhile, could watch a whole array of connected clients and pounce on the few sockets that were actually ready to send or receive data at any given moment? The result would be an event-driven server that sits in a tight loop watching many clients; I have written an example, shown in Listing 7–7.

Listing 7–7. *A Non-blocking Event-Driven Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - server_poll.py
# An event-driven approach to serving several clients with poll().

import launcelot
import select

listen_sock = launcelot.setup()
sockets = { listen_sock.fileno(): listen_sock }
requests = {}
responses = {}

poll = select.poll()
poll.register(listen_sock, select.POLLIN)

while True:
    for fd, event in poll.poll():
        sock = sockets[fd]
        # Removed closed sockets from our list.
        if event & (select.POLLHUP | select.POLLERR | select.POLLNVAL):
            poll.unregister(fd)
            del sockets[fd]
            requests.pop(sock, None)
            responses.pop(sock, None)

        # Accept connections from new sockets.
        elif sock is listen_sock:
            newsock, sockname = sock.accept()
            newsock.setblocking(False)
            fd = newsock.fileno()
            sockets[fd] = newsock
            poll.register(fd, select.POLLIN)
            requests[newsock] = ''

        # Collect incoming data until it forms a question.
        elif event & select.POLLIN:
            data = sock.recv(4096)
            if not data:      # end-of-file
                sock.close()  # makes POLLNVAL happen next time
                continue
            requests[sock] += data
            if '?' in requests[sock]:
                question = requests.pop(sock)
                answer = dict(launcelot.qa)[question]
                poll.modify(sock, select.POLLOUT)
                responses[sock] = answer
```

```
»   »   # Send out pieces of each reply until they are all sent.
»   »   elif event & select.POLLOUT:
»   »   »   response = responses.pop(sock)
»   »   »   n = sock.send(response)
»   »   »   if n < len(response):
»   »   »   »   responses[sock] = response[n:]
»   »   »   else:
»   »   »   »   poll.modify(sock, select.POLLIN)
»   »   »   »   requests[sock] = ''
```

The main loop in this program is controlled by the `poll` object, which is queried at the top of every iteration. The `poll()` call is a blocking call, just like the `recv()` call in our simple server; so the difference is *not* that our first server used a blocking operating system call and that this second server is somehow avoiding that. No, this server blocks too; the difference is that `recv()` has to wait on one single client, while `poll()` can wait on dozens or hundreds of clients, and return when any of them shows activity.

You can see that everywhere that the original server had exactly one of something—one client socket, one question string, or one answer ready to send—this event-driven server has to keep entire arrays or dictionaries, because it is like a poker dealer who has to keep cards flying to all of the players at once.

The way `poll()` works is that we tell it which sockets we need to monitor, and whether each socket interests us because we want to read from it or write to it. When one or more of the sockets are ready, `poll()` returns and provides a list of the sockets that we can now use.

To keep things straight when reading the code, think about the lifespan of one particular client and trace what happens to its socket and data.

1. The client will first do a `connect()`, and the server's `poll()` call will return and declare that there is data ready on the main listening socket. That can mean only one thing, since—as we learned in Chapter 3—actual data never appears on a stream socket that is being used to `listen()`: it means that a new client has connected. So we `accept()` the connection and tell our `poll` object that we want to be notified when data becomes available for reading from the new socket. To make sure that the `recv()` and `send()` methods on the socket never block and freeze our event loop, we call the `setblocking()` socket method with the value `False` (which means "blocking is *not* allowed").

2. When data becomes available, the incoming string is appended to whatever is already in the `requests` dictionary under the entry for that socket. (Yes, sockets can safely be used as dictionary keys in Python!)

3. We keep accepting more data until we see a question mark, at which point the Launcelot question is complete. The questions are so short that, in practice, they probably all arrive in the very first `recv()` from each socket; but just to be safe, we have to be prepared to make several `recv()` calls until the whole question has arrived. We then look up the appropriate answer, store it in the `responses` dictionary under the entry for this client socket, and tell the `poll` object that we no longer want to listen for more data from this client but instead want to be told when its socket can start accepting outgoing data.

4. Once a socket is ready for writing, we send as much of the answer as will fit into one `send()` call on the client socket. This, by the way, is a big reason `send()` returns a length: because if you use it in non-blocking mode, then it might be able to send only some of your bytes without making you wait for a buffer to drain back down.

5. Once this server has finished transmitting the answer, we tell the poll object to swap the client socket back over to being listened to for new incoming data.

6. After many question-answer exchanges, the client will finally close the connection. Oddly enough, the POLLHUP, POLLERR, and POLLNVAL circumstances that poll() can tell us about—all of which indicate that the connection has closed one way or another—are returned only if we are trying to write to the socket, not read from it. So when an attempt to read returns zero bytes, we have to tell the poll object that we now want to write to the socket so that we receive the official notification that the connection is closed.

The performance of this vastly improved server design is shown in Figure 7–2. By the time its throughput begins to really degrade, it has achieved twice the requests per second of the simple server with which we started the chapter.

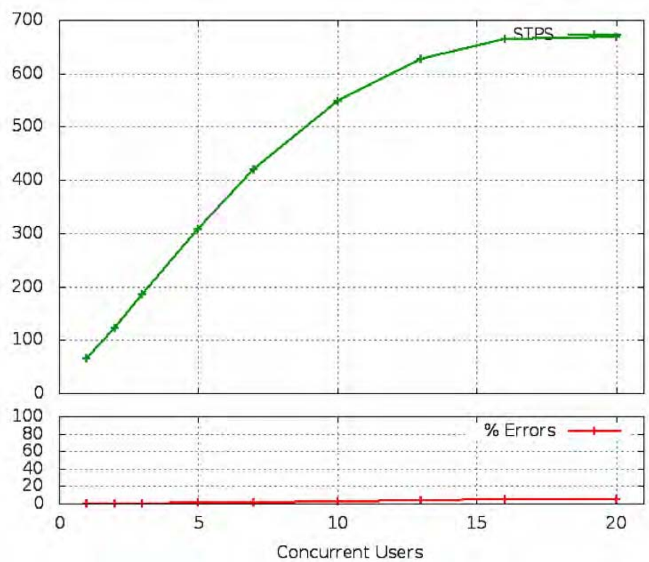


Figure 7–2. *Polling server benchmark*

Of course, this factor-of-two improvement is very specific to the design of this server and the particular memory and processor layout of my laptop. Depending on the length and cost of client requests, other network services could see much more or much less improvement than our Launcelot service has displayed here. But you can see that a pure event-driven design like this one turns the focus of your program away from the question of what one particular client will want next, and toward the question of what event is ready to happen regardless of where it comes from.

Poll vs. Select

A slightly older mechanism for writing event-driven servers that listen to sockets is to use the select() call, which like poll() is available from the Python select module in the Standard Library. I chose to use

poll() because it produces much cleaner code, but many people choose select() because it is supported on Windows.

As servers today are being asked to support greater and greater numbers of clients, some people have abandoned both select() and poll() and have opted for the epoll() mechanism provided by Linux or the kqueue() call under BSD. Some programmers have made this switch with solid numbers to back them up; other developers seem to switch simply because the latter calls are newer, but never actually check whether they will improve performance in their specific case.

Which mechanism should you use in your own code?

My advice is actually to avoid both of them! In my opinion, unless you have very specialized needs, you are not using your time well if you are sitting down and writing anything that looks like Listing 7–7. It is very difficult to get such code right—you will note that I myself did not include any real error handling, because otherwise the code would have become well-nigh unreadable, and the point of the listing is just to introduce the concept.

Instead of sitting down with W. Richard Stevens's *Advanced Programming in the UNIX Environment* and the manual pages for your operating system and trying to puzzle out exactly how to use select() or poll() with correct attention to all of the edge cases on your particular platform, you should be using an event-driven framework that does the work for you.

But we will look at frameworks in a moment; first, we need to get some terminology straight.

The Semantics of Non-blocking

I should add a quick note about how recv() and send() behave in non-blocking mode, when you have called setblocking(False) on their socket. A poll() loop like the one just shown means that we never wind up calling either of these functions when they cannot accept or provide data. But what if we find ourselves in a situation where we want to call either function in non-blocking mode and do not yet know whether the socket is ready?

For the recv() call, these are the rules:

- If data is ready, it is returned.
- If no data has arrived, socket.error is raised.
- If the connection has closed, '' is returned.

This behavior might surprise you: a closed connection returns a value, but a still-open connection raises an exception. The logic behind this behavior is that the first and last possibilities are both possible in blocking mode as well: either you get data back, or finally the connection closes and you get back an empty string. So to communicate the extra, third possibility that can happen in non-blocking mode—that the connection is still open but no data is ready yet—an exception is used.

The behavior of non-blocking send() is similar:

- Some data is sent, and its length is returned.
- The socket buffers are full, so socket.error is raised.
- If the connection is closed, socket.error is also raised.

This last possibility may introduce a corner case that Listing 7–7 does not attempt to detect: that poll() could say that a socket is ready for sending, but a FIN packet from the client could arrive right after the server is released from its poll() but before it can start up its send() call.

Event-Driven Servers Are Blocking and Synchronous

The terminology surrounding event-driven servers like the one shown in Listing 7–7 has become quite tangled. Some people call them "non-blocking," despite the fact that the `poll()` call blocks, and others call them "asynchronous" despite the fact that the program executes its statements in their usual linear order. How can we keep these claims straight?

First, I note that everyone seems to agree that it is correct to call such a server "event-driven," which is why I am using that term here.

Second, I think that when people loosely call these systems "non-blocking," they mean that it does not block waiting for any *particular* client. The calls to send and receive data on any one socket are not allowed to pause the entire server process. But in this context, the term "non-blocking" has to be used very carefully, because back in the old days, people wrote programs that indeed did not block on *any* calls, but instead entered a "busy loop" that repeatedly polled a machine's I/O ports watching for data to arrive. That was fine if your program was the only one running on the machine; but such programs are a disaster when run under modern operating systems. The fact that event-driven servers can choose to block with `select()` or `poll()` is the very reason they can function as efficient services on the machine, instead of being resource hogs that push CPU usage immediately up to 100%.

Finally, the term "asynchronous" is a troubled one. At least on Unix systems, it was traditionally reserved for programs that interacted with their environment by receiving signals, which are violent interruptions that yank your program away from whatever statement it is executing and run special signal-handling code instead. Check out the `signal` module in the Standard Library for a look at how Python can hook into this mechanism. Programs that could survive having any part of their code randomly interrupted were rather tricky to write, and so asynchronous programming was quite correctly approached with great caution. And at bottom, computers themselves are inherently asynchronous. While your operating system does not receive "signals," which are a concept invented for user-level programs, they do receive IRQs and other hardware interrupts. The operating system has to have handlers ready that will correctly respond to each event without disturbing the code that will resume when the handler is complete.

So it seems to me that enough programming is really asynchronous, even today, that the term should most properly be reserved for the "hard asynchrony" displayed by IRQs and signal handlers. But, on the other hand, one must admit that while the program statements in Listing 7–7 are synchronous with respect to one another—they happen one right after the other, without surprises, as in any Python program—the I/O itself does *not* arrive in order. You might get a string from one client, then have to finish sending an answer to a second client, then suddenly find that a third client has hung up its connection. So we can grudgingly admit that there is a "soft asynchrony" here that involves the fact that network operations happen whenever they want, instead of happening lockstep in some particular order.

So in peculiar and restricted senses, I believe, an event-driven server can indeed be called non-blocking and asynchronous. But those terms can also have much stronger meanings that certainly do not apply to Listing 7–7, so I recommend that we limit ourselves to the term "event-driven" when we talk about it.

Twisted Python

I mentioned earlier that you are probably doing something wrong if you are sitting down to wrestle with `select()` or `poll()` for any reason other than to write a new event-driven framework. You should normally treat them as low-level implementation details that you are happy to know about—having seen and studied Listing 7–7 makes you a wiser person, after all—but that you also normally leave to others. In the same way, understanding the UTF-8 string encoding is useful, but sitting down to write your own encoder in Python is probably a sign that you are re-inventing a wheel.

Now it happens that Python comes with an event-driven framework built into the Standard Library, and you might think that the next step would be for me to describe it. In fact, I am going to recommend that you ignore it entirely! It is a pair of ancient modules, asyncore and asynchat, that date from the early days of Python—you will note that all of the classes they define are lowercase, in defiance of both good taste and all subsequent practice—and that they are difficult to use correctly. Even with the help of Doug Hellmann's "Python Module of the Week" post about each of them, it took me more than an hour to write a working example of even our dead-simple Launcelot protocol.

If you are curious about these old parts of the Standard Library, then download the source bundle for this book and look for the program in the Chapter 7 directory named server_async.py, which is the result of my one foray into asyncore programming. But here in the book's text, I shall say no more about them.

Instead, we will talk about Twisted Python.

Twisted Python is not simply a framework; it is almost something of a movement. In the same way that Zope people have their own ways of approaching Python programming, the Twisted community has developed a way of writing Python that is all their own. Take a look at Listing 7–8 for how simple our event-driven server can become if we leave the trouble of dealing with the low-level operating system calls to someone else.

Listing 7–8. *Implementing Launcelot in Twisted*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - server_twisted.py
# Using Twisted to serve Launcelot users.

from twisted.internet.protocol import Protocol, ServerFactory
from twisted.internet import reactor
import launcelot

class Launcelot(Protocol):
»   def connectionMade(self):
»   »   self.question = ''

»   def dataReceived(self, data):
»   »   self.question += data
»   »   if self.question.endswith('?'):
»   »   »   self.transport.write(dict(launcelot.qa)[self.question])
»   »   »   self.question = ''

factory = ServerFactory()
factory.protocol = Launcelot
reactor.listenTCP(1060, factory)
reactor.run()
```

Since you have seen Listing 7–7, of course, you know what Twisted must be doing under the hood: it must use select() or poll() or epoll()—and the glory of the approach is that we do not really care which—and then instantiate our Launcelot class once for every client that connects. From then on, every event on that socket is translated into a method call to our object, letting us write code that appears to be thinking about just one client at a time. But thanks to the fact that Twisted will create dozens or hundreds of our Launcelot protocol objects, one corresponding to each connected client, the result is an event loop that can respond to whichever client sockets are ready.

It is clear in Listing 7–8 that we are accumulating data in a way that keeps the event loop running; after all, dataReceived() always returns immediately while it is still accumulating the full question string. But what stops the server from blocking when we call the write() method of our data transport? The answer is that write() does not actually attempt any immediate socket operation; instead, it schedules

the data to be written out by the event loop as soon as the client socket is ready for it, exactly as we did in our own event-driven loop.

There are more methods available on a Twisted `Protocol` class than we are using here—methods that are called when a connection is made, when it is closed, when it closes unexpectedly, and so forth. Consult their documentation to learn all of your options.

The performance of Twisted, as you can see from Figure 7–3, is somewhat lower than that of our handwritten event loop, but, of course, it is doing a lot more work. And if we actually padded out our earlier loop to include all of the error handling and compatibility that are supported by Twisted, then the margin would be closer.

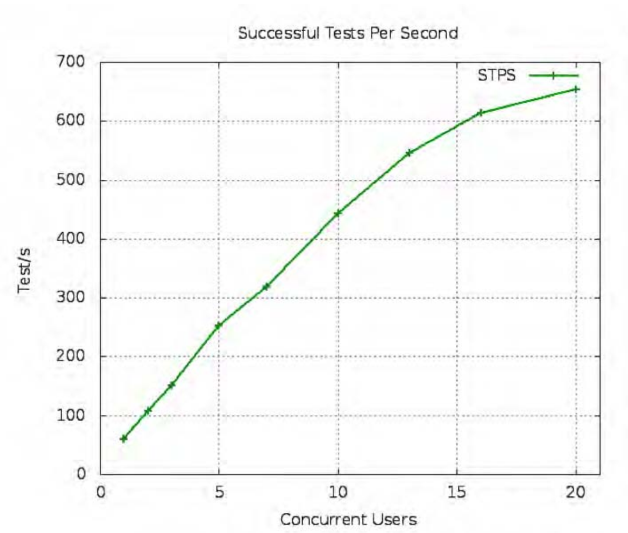


Figure 7–3. *Twisted server benchmark*

The real magic of Twisted—which we lack the space to explore here—happens when you write protocols that have to speak to several partners at once rather than just one. Our Launcelot service can generate each reply immediately, by simply looking in a dictionary; but what if generating an answer involved reading from disk, or querying another network service, or talking to a local database?

When you have to invoke an operation that actually takes time, Twisted lets you provide it with one or more callback functions that it calls *deferreds*. And this is really the art of writing with Twisted: the craft of putting together short and long series of deferred functions so that, as blocks of data roll in from the disk or replies come back from a database server, all the right functions fire to construct an answer and get it delivered back to your client. Error handling becomes the practice of making sure that appropriate error callbacks are always available in case any particular network or I/O operation fails.

Some Python programmers find deferreds to be an awkward pattern and prefer to use other mechanisms when they need to serve many network clients at once; the rest of this chapter is dedicated to them. But if the idea of chaining callback functions intrigues you or seems to fit your mind—or if you simply want to benefit from the long list of protocols that Twisted has already implemented, and from the community that has gathered around it—then you might want to head off to the Twisted web site and try tackling its famous tutorial: `http://twistedmatrix.com/documents/current/core/howto/tutorial/`.

I myself have never based a project on Twisted because deferreds always make me feel as though I am writing my program backward; but many people find it quite pleasant once they are used to it.

Load Balancing and Proxies

Event-driven servers take a single process and thread of control and make it serve as many clients as it possibly can; once every moment of its time is being spent on clients that are ready for data, a process really can do no more. But what if one thread of control is simply not enough for the load your network service needs to meet?

The answer, obviously, is to run several instances of your service and to distribute clients among them. This requires a key piece of software: a *load balancer* that runs on the port to which all of the clients will be connecting, and which then turns around and gives each of the running instances of your service the data being sent by some fraction of the incoming clients. The load balancer thus serves as a *proxy*: to network clients it looks like your server, but to your server it looks like a client, and often neither side knows the proxy is even there.

Load balancers are such critical pieces of infrastructure that they are often built directly into network hardware, like that sold by Cisco, Barracuda, and f5. On a normal Linux system, you can run software like HAProxy or delve into the operating system's firewall rules and construct quite efficient load balancing using the Linux Virtual Server (LVS) subsystem.

In the old days, it was common to spread load by simply giving a single domain name several different IP addresses; clients looking up the name would be spread randomly across the various server machines. The problem with this, of course, is that clients suffer when the server to which they are assigned goes down; modern load balancers, by contrast, can often recover when a back-end server goes down by moving its live connections over to another server without the client even knowing.

The one area in which DNS has retained its foothold as a load-balancing mechanism is geography. The largest service providers on the Internet often resolve hostnames to different IP addresses depending on the continent, country, and region from which a particular client request originates. This allows them to direct traffic to server rooms that are within a few hundred miles of each customer, rather than requiring their connections to cross the long and busy data links between continents.

So why am I mentioning all of these possibilities *before* tackling the ways that you can move beyond a single thread of control on a single machine with threads and processes?

The answer is that I believe load balancing should be considered up front in the design of any network service because it is the only approach that really scales. True, you can buy servers these days of more than a dozen cores, mounted in machines that support massive network channels; but if, someday, your service finally outgrows a single box, then you will wind up doing load balancing. And if load balancing can help you distribute load between entirely *different* machines, why not also use it to help you keep several copies of your server active on the *same* machine?

Threading and forking, it turns out, are merely limited special cases of load balancing. They take advantage of the fact that the operating system will load-balance incoming connections among all of the threads or processes that are running `accept()` against a particular socket. But if you are going to have to run a separate load balancer in front of your service anyway, then why go to the trouble of threading or forking on each individual machine? Why not just run 20 copies of your simple single-threaded server on 20 different ports, and then list them in the load balancer's configuration?

Of course, you might know ahead of time that your service will never expand to run on several machines, and might want the simplicity of running a single piece of software that can by itself use several processor cores effectively to answer client requests. But you should keep in mind that a multi-threaded or multi-process application is, within a single piece of software, doing what might more cleanly be done by configuring a proxy standing outside your server code.

Threading and Multi-processing

The essential idea of a threaded or multi-process server is that we take the simple and straightforward server that we started out with—the one way back in Listing 7–2, the one that waits repeatedly on a

single client and then sends back the information it needs—and run several copies of it at once so that we can serve several clients at once, without making them wait on each other.

The event-driven approaches in Listings 7–7 and 7–8 place upon our own program the burden of figuring out which client is ready next, and how to interleave requests and responses depending on the order in which they arrive. But when using threads and processes, you get to transfer this burden to the operating system itself. Each thread controls one client socket; it can use blocking `recv()` and `send()` calls to wait until data can be received and transmitted; and the operating system then decides which workers to leave idle and which to wake up.

Using multiple threads or processes is very common, especially in high-capacity web and database servers. The Apache web server even comes with both: its `prefork` module offers a pool of processes, while the `worker` module runs multiple threads instead.

Listing 7–9 shows a simple server that creates multiple workers. Note how pleasantly symmetrical the Standard Library authors have made the interface between threads and processes, thanks especially to Jesse Noller and his recent work on the `multiprocessing` module. The main program logic does not even know which solution is being used; the two classes have a similar enough interface that either `Thread` or `Process` can here be used interchangeably.

Listing 7–9. *Multi-threaded or Multi-process Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - server_multi.py
# Using multiple threads or processes to serve several clients in parallel.

import sys, time, launcelot
from multiprocessing import Process
from server_simple import server_loop
from threading import Thread

WORKER_CLASSES = {'thread': Thread, 'process': Process}
WORKER_MAX = 10

def start_worker(Worker, listen_sock):
    worker = Worker(target=server_loop, args=(listen_sock,))
    worker.daemon = True  # exit when the main process does
    worker.start()
    return worker

if __name__ == '__main__':
    if len(sys.argv) != 3 or sys.argv[2] not in WORKER_CLASSES:
        print >>sys.stderr, 'usage: server_multi.py interface thread|process'
        sys.exit(2)
    Worker = WORKER_CLASSES[sys.argv.pop()]  # setup() wants len(argv)==2

    # Every worker will accept() forever on the same listening socket.

    listen_sock = launcelot.setup()
    workers = []
    for i in range(WORKER_MAX):
        workers.append(start_worker(Worker, listen_sock))

    # Check every two seconds for dead workers, and replace them.

    while True:
        time.sleep(2)
```

```
»	»	for worker in workers:
»	»	»	if not worker.is_alive():
»	»	»	»	print worker.name, "died; starting replacement"
»	»	»	»	workers.remove(worker)
»	»	»	»	workers.append(start_worker(Worker, listen_sock))
```

First, notice how this server is able to re-use the simple, procedural approach to answering client requests that it imports from the `launcelot.py` file we introduced in Listing 7–2. Because the operating system keeps our threads or processes separate, they do not have to be written with any awareness that other workers might be operating at the same time.

Second, note how much work the operating system is doing for us! It is letting multiple threads or processes all call `accept()` on the very same server socket, and instead of raising an error and insisting that only one thread at a time be able to wait for an incoming connection, the operating system patiently queues up all of our waiting workers and then wakes up one worker for each new connection that arrives. The fact that a listening socket can be shared at all between threads and processes, and that the operating system does round-robin balancing among the workers that are waiting on an `accept()` call, is one of the great glories of the POSIX network stack and execution model; it makes programs like this very simple to write.

Third, although I chose not to complicate this listing with error-handling or logging code—any exceptions encountered in a thread or process will be printed as tracebacks directly to the screen—I did at least throw in a loop in the master thread that checks the health of the workers every few seconds, and starts up replacement workers for any that have failed.

Figure 7–4 shows the result of our efforts: performance that is far above that of the single-threaded server, and that also beats slightly both of the event-driven servers we looked at earlier.

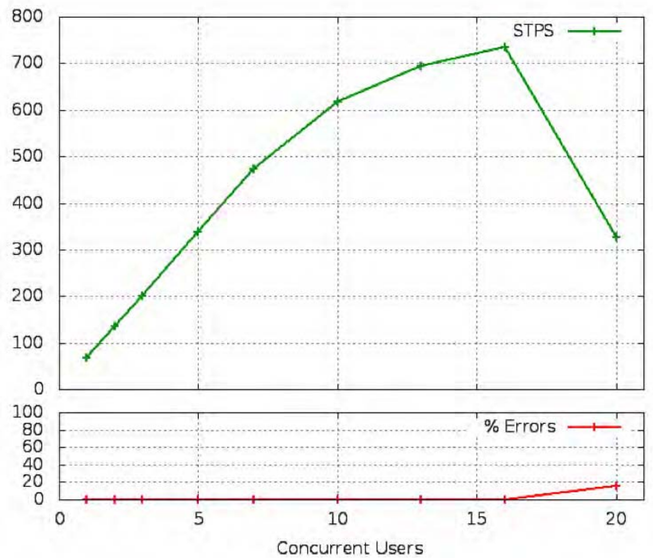


Figure 7–4. *Multi-process server benchmark*

Again, given the limitations of my small duo-core laptop, the server starts falling away from linear behavior as the load increases from 5 to 10 simultaneous clients, and by the time it reaches 15 concurrent users, the number of 10-question request sequences that it can answer every second has fallen from around 70 per client to less than 50. And then—as will be familiar to anyone who has studied

queuing theory, or run benchmarks like this before—its performance goes tumbling off of a cliff as the expense of trying to serve so many clients at once finally starts to overwhelm its ability to get any work done.

Note that running threads under standard C Python will impose on your server the usual limitation that no more than one thread can be running Python code at any given time. Other implementations, like Jython and IronPython, avoid this problem by building on virtual machine runtimes that lock individual data structures to protect them from simultaneous access by several threads at once. But C Python has no better approach to concurrency than to lock the entire Python interpreter with its Global Interpreter Lock (GIL), and then release it again when the code reaches a call like `accept()`, `recv()`, or `send()` that might wait on external I/O.

How many children should you run? This can be determined only by experimentation against your server on the particular machine that will be running it. The number of server cores, the speed or slowness of the clients that will be connecting, and even the speed of your RAM bus can affect the optimum number of workers. I recommend running a series of benchmarks with varying numbers of workers, and seeing which configuration seems to give you the best performance.

Oh—and, one last note: the `multiprocessing` module does a good job of cleaning up your worker processes if you exit from it normally or kill it softly from the console with Ctrl+C. But if you kill the main process with a signal, then the children will be orphaned and you will have to kill them all individually. The worker processes are normally children of the parent (here I have briefly changed `WORKER_MAX` to `3` to reduce the amount of output):

```
$ python server_multi.py localhost process
$ ps f|grep 'python server_[m]ulti'
11218 pts/2    S+     0:00  \_ python server_multi.py localhost process
11219 pts/2    S+     0:00      \_ python server_multi.py localhost process
11220 pts/2    S+     0:00      \_ python server_multi.py localhost process
11221 pts/2    S+     0:00      \_ python server_multi.py localhost process
```

Running `ps` on a POSIX machine with the `f` option shows processes as a family tree, with parents above their children. And I randomly added square brackets to the `m` in the `grep` pattern so that the pattern does not match itself; it is always annoying when you `grep` for some particular process, and the `grep` process also gets returned because the pattern matches itself.

If I violently `kill` the parent, then unfortunately all three children remain running, which not only is annoying but also stops me from re-running the server since the children continue to hold open the listening socket:

```
$ kill 11218
$ ps f|grep 'python server_[m]ulti'
11228 pts/2    S      0:00 python server_multi.py localhost process
11227 pts/2    S      0:00 python server_multi.py localhost process
11226 pts/2    S      0:00 python server_multi.py localhost process
```

So manually killing them is the only recourse, with something like this:

```
$ kill $(ps f|grep 'python server_[m]ulti'|awk '{print$1}')
```

If you are concerned enough about this problem with the `multiprocessing` module, then look on the Web for advice about how to use signal handling (the `kill` command operates by sending a signal, which the parent process is failing to intercept) to catch the termination signal and shut down the workers.

Threading and Multi-processing Frameworks

As usual, many programmers prefer to let someone else worry about the creation and maintenance of their worker pool. While the `multiprocessing` module does have a `Pool` object that will distribute work to

several child processes (and it is rumored to also have an undocumented ThreadPool), that mechanism seems focused on distributing work from the master thread rather than on accepting different client connections from a common listening socket. So my last example in this chapter will be built atop the modest SocketServer module in the Python Standard Library.

The SocketServer module was written a decade ago, which is probably obvious in the way it uses multiclassing and mix-ins—today, we would be more likely to use dependency injection and pass in the threading or forking engine as an argument during instantiation. But the arrangement works well enough; in Listing 7–10, you can see how small our multi-threaded server becomes when it takes advantage of this framework. (There is also a ForkingMixIn that you can use if you want it to spawn several processes—at least on a POSIX system.)

Listing 7–10. *Using the Standard Library Socket Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - server_SocketServer.py
# Answering Launcelot requests with a SocketServer.

from SocketServer import ThreadingMixIn, TCPServer, BaseRequestHandler
import launcelot, server_simple, socket

class MyHandler(BaseRequestHandler):
»   def handle(self):
»   »   server_simple.handle_client(self.request)

class MyServer(ThreadingMixIn, TCPServer):
»   allow_reuse_address = 1
»   # address_family = socket.AF_INET6  # if you need IPv6

server = MyServer(('', launcelot.PORT), MyHandler)
server.serve_forever()
```

Note that this framework takes the opposite tack to the server that we built by hand in the previous section. Whereas our earlier example created the workers up front so that they were all sharing the same listening socket, the SocketServer does all of its listening in the main thread and creates one worker each time accept() returns a new client socket. This means that each request will run a bit more slowly, since the client has to wait for the process or thread to be created before it can receive its first answer; and this is evident in Figure 7–5, where the volume of requests answered runs a bit lower than it did in Figure 7–4.

A disadvantage of the SocketServer classes, so far as I can see, is that there is nothing to stop a sudden flood of client connections from convincing the server to spin up an equal number of threads or processes—and if that number is large, then your computer might well slow to a crawl or run out of resources rather than respond constructively to the demand. Another advantage to the design of Listing 7–9, then, is that it chooses ahead of time how many simultaneous requests can usefully be underway, and leaves additional clients waiting for an accept() on their connections before they can start contributing to the load on the server.

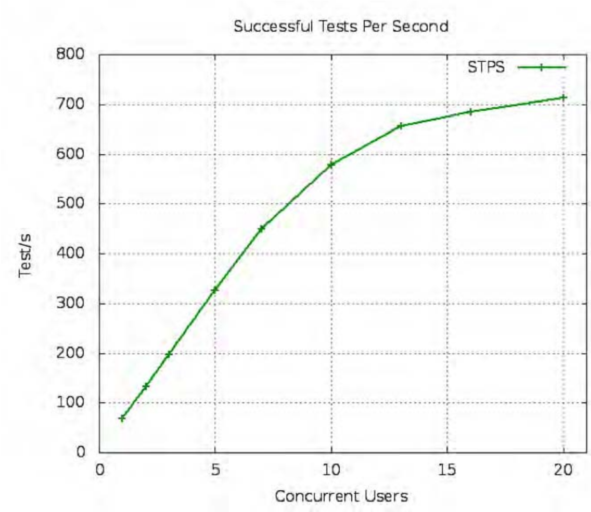


Figure 7–5. *Multi-process server benchmark*

Process and Thread Coordination

I have written this chapter with the idea that each client request you want to handle can be processed independently, and without making the thread or process that is answering it share any in-memory data structures with the rest of your threads.

This means that if you are connecting to a database from your various worker threads, I assume that you are using a thread-safe database API. If your workers need to read from disk or update a file, than I assume that you are doing so in a way that will be safe if two, or three, or four threads all try using the same resource at once.

But if this assumption is wrong—if you want the various threads of control in your application to share data, update common data structures, or try to send messages to each other—then you have far deeper problems than can be solved in a book on network programming. You are embarking, instead, on an entire discipline of its own known as "concurrent programming," and will have to either restrict yourself to tools and methodologies that make concurrency safe, or be fiendishly clever with low-level mechanisms like locks, semaphores, and condition variables.

I have four pieces of advice if you think that you will take this direction.

First, make sure that you have the difference between threads and processes clear in your head. Listing 7–9 treated the two mechanisms as equivalent because it was not trying to maintain any shared data structures that the workers would have to access. But if your workers need to talk to one another, then threads let them do so in-memory—any global variables in each module, as well as changes to such variables, will be immediately visible to all other threads—whereas multiple processes can share only data structures that you explicitly create for sharing using the special mechanisms inside the `multiprocessing` module. On the one hand, this makes threading more convenient since data is shared by default. On the other hand, this makes processes far more safe, since you explicitly have to opt-in each data structure to being shared, and cannot get bitten by state that gets shared accidentally.

Second, use high-level data structures whenever possible. Python provides queues, for example, that can operate either between normal threads (from the `queue` module) or between processes (see the `multiprocessing` module). Passing data back and forth with these well-designed tools is far less

complicated than trying to use locks and semaphores on your own to signal when data is ready to be consumed.

Third, limit your use of shared data to small and easily protected pieces of code. Under no circumstances should you be spreading primitive semaphores and condition variables across your entire code base and hope that the collective mass that results will somehow operate correctly and without deadlocks or data corruption. Choose a few conceptually small points of synchronization where the gears of your program will mesh together, and do your hard thinking there in one place to make sure that your program will operate correctly.

Finally, look very hard at both the Standard Library and the Package Index for evidence that some other programmer before you has faced the data pattern you are trying to implement and has already taken the time to get it right. Well-maintained public projects with several users are fun to build on, because their users will already have run into many of the situations where they break long before you are likely to run into these situations in your own testing.

But most network services are not in this class. Examine, for instance, the construction of most view functions or classes in a typical Python web framework: they manipulate the parameters that have been passed in to produce an answer, without knowing anything about the other requests that other views are processing at the same time. If they need to share information or data with the other threads or processes running the same web site, they use a hardened industrial tool like a database to maintain their shared state in a way that all of their threads can get to without having to manage their own contention. That, I believe, is the way to go about writing network services: write code that concerns itself with local variables and local effects, and that leaves all of the issues of locking and concurrency to people like database designers that are good at that sort of thing.

Running Inside inetd

For old times' sake, I should not close this chapter without mentioning `inetd`, a server used long ago on Unix systems to avoid the expense of running several Internet daemons. Back then, the RAM used by each running process was a substantial annoyance. Today, of course, even the Ubuntu laptop on which I am typing is running dozens of services just to power things like the weather widget in my toolbar, and the machine's response time seems downright snappy despite running—let's see—wow, 229 separate processes all at the same time. (Yes, I know, that count includes one process for each open tab in Google Chrome.)

So the idea was to have an `/etc/inetd.conf` file where you could list all of the services you wanted to provide, along with the name of the program that should be run to answer each request. Thus, `inetd` took on the job of opening every one of those ports; using `select()` or `poll()` to watch all of them for incoming client connections; and then calling `accept()` and handing the new client socket off to a new copy of the process named in the configuration file.

Not only did this arrangement save time and memory on machines with many lightly used services, but it became an important step in securing a machine once Wietse Venema invented the TCP Wrappers (see Chapter 6). Suddenly everyone was rewriting their `inetd.conf` files to call Wietse's access-control code before actually letting each raw service run. The configuration files had looked like this:

```
ftp    stream  tcp     nowait  root    in.ftpd  in.ftpd -l -a
telnet stream  tcp     nowait  root    in.telnetd  in.telnetd
talk   dgram   udp     wait    nobody  in.talkd  in.talkd
finger stream  tcp     nowait  nobody  in.fingerd  in.fingerd
```

Once Wietse's `tcpd` binary was installed, the `inetd.conf` file would be rewritten like this:

```
ftp    stream  tcp     nowait  root    /usr/sbin/tcpd  in.ftpd -l -a
telnet stream  tcp     nowait  root    /usr/sbin/tcpd  in.telnetd
talk   dgram   udp     wait    nobody  /usr/sbin/tcpd  in.talkd
finger stream  tcp     nowait  nobody  /usr/sbin/tcpd  in.fingerd
```

The tcpd binary would read the /etc/hosts.allow and hosts.deny files and enforce any access rules it found there—and also possibly log the incoming connection—before deciding to pass control through to the actual service being protected.

If you are writing a Python service to be run from inetd, the client socket returned by the inetd accept() call will be passed in as your standard input and output. If you are willing to have standard file buffering in between you and your client—and to endure the constant requirement that you flush() the output every time that you are ready for the client to receive your newest block of data—then you can simply read from standard input and write to the standard output normally. If instead you want to run real send() and recv() calls, then you will have to convert one of your input streams into a socket and then close the originals (because of a peculiarity of the Python socket fromfd() call: it calls dup() before handing you the socket so that you can close the socket and file descriptor separately):

```
import socket, sys
sock = socket.fromfd(sys.stdin.fileno(), socket.AF_INET, socket.SOCK_STREAM)
sys.stdin.close()
```

In this sense, inetd is very much like the CGI mechanism for web services: it runs a separate process for every request that arrives, and hands that program the client socket as though the program had been run with a normal standard input and output.

Summary

Network servers typically need to run as daemons so that they do not exit when a particular user logs out, and since they will have no controlling terminal, they will need to log their activity to files so that administrators can monitor and debug them. Either supervisor or the daemon module is a good solution for the first problem, and the standard logging module should be your focus for achieving the second.

One approach to network programming is to write an event-driven program, or use an event-driven framework like Twisted Python. In both cases, the program returns repeatedly to an operating system–supported call like select() or poll() that lets the server watch dozens or hundreds of client sockets for activity, so that you can send answers to the clients that need it while leaving the other connections idle until another request is received from them.

The other approach is to use threads or processes. These let you take code that knows how to talk to one client at a time, and run many copies of it at once so that all connected clients have an agent waiting for their next request and ready to answer it. Threads are a weak solution under C Python because the Global Interpreter Lock prevents any two of them from both running Python code at the same time; but, on the other hand, processes are a bit larger, more expensive, and difficult to manage.

If you want your processes or threads to communicate with each other, you will have to enter the rarefied atmosphere of concurrent programming, and carefully choose mechanisms that let the various parts of your program communicate with the least chance of your getting something wrong and letting them deadlock or corrupt common data structures. Using high-level libraries and data structures, where they are available, is always far preferable to playing with low-level synchronization primitives yourself.

In ancient times, people ran network services through inetd, which hands each server an already-accepted client connection as its standard input and output. Should you need to participate in this bizarre system, be prepared to turn your standard file descriptors into sockets so that you can run real socket methods on them.

CHAPTER 8

Caches, Message Queues, and Map-Reduce

This chapter, though brief, might be one of the most important in this book. It surveys the handful of technologies that have together become fundamental building blocks for expanding applications to Internet scale.

In the following pages, this book reaches its turning point. The previous chapters have explored the sockets API and how Python can use the primitive IP network operations to build communication channels. All of the subsequent chapters, as you will see if you peek ahead, are about very particular protocols built atop sockets—about how to fetch web documents, send e-mails, and connect to server command lines.

What sets apart the tools that we will be looking at here? They have several characteristics:

- Each of these technologies is popular because it is a powerful tool. The point of using Memcached or a message queue is that it is a very well-written service that will solve a particular problem for you—not because it implements an interesting protocol that different organizations are likely to use to communicate.
- The problems solved by these tools tend to be internal to an organization. You often cannot tell from outside which caches, queues, and load distribution tools are being used to power a particular web site.
- While protocols like HTTP and SMTP were built with specific payloads in mind—hypertext documents and e-mail messages, respectively—caches and message queues tend to be completely agnostic about the data that they carry for you.

This chapter is not intended to be a manual for any of these technologies, nor will code examples be plentiful. Ample documentation for each of the libraries mentioned exists online, and for the more popular ones, you can even find entire books that have been written about them. Instead, this chapter's purpose is to introduce you to the problem that each tool solves; explain how to use the service to address that issue; and give a few hints about using the tool from Python.

After all, the greatest challenge that a programmer often faces—aside from the basic, lifelong process of learning to program itself—is knowing that a solution exists. We are inveterate inventors of wheels that already exist, had we only known it. Think of this chapter as offering you a few wheels in the hopes that you can avoid hewing them yourself.

Using Memcached

Memcached is the "memory cache daemon." Its impact on many large Internet services has been, by all accounts, revolutionary. After glancing at how to use it from Python, we will discuss its implementation, which will teach us about a very important modern network concept called *sharding*.

The actual procedures for using Memcached are designed to be very simple:

- You run a Memcached daemon on every server with some spare memory.
- You make a list of the IP address and port numbers of your new Memcached daemons, and distribute this list to all of the clients that will be using the cache.
- Your client programs now have access to an organization-wide blazing-fast key-value cache that acts something like a big Python dictionary that all of your servers can share. The cache operates on an LRU (least-recently-used) basis, dropping old items that have not been accessed for a while so that it has room to both accept new entries and keep records that are being frequently accessed.

Enough Python clients are currently listed for Memcached that I had better just send you to the page that lists them, rather than try to review them here: `http://code.google.com/p/memcached/wiki/Clients`.

The client that they list first is written in pure Python, and therefore will not need to compile against any libraries. It should install quite cleanly into a virtual environment (see Chapter 1), thanks to being available on the Python Package Index:

```
$ pip install python-memcached
```

The interface is straightforward. Though you might have expected an interface that more strongly resembles a Python dictionary with native methods like `__getitem__`, the author of python-memcached chose instead to use the same method names as are used in other languages supported by Memcached—which I think was a good decision, since it makes it easier to translate Memcached examples into Python:

```
>>> import memcache
>>> mc = memcache.Client(['127.0.0.1:11211'])
>>> mc.set('user:19', '{name: "Lancelot", quest: "Grail"}')
True
>>> mc.get('user:19')
'{name: "Lancelot", quest: "Grail"}'
```

The basic pattern by which Memcached is used from Python is shown in Listing 8–1. Before embarking on an (artificially) expensive operation, it checks Memcached to see whether the answer is already present. If so, then the answer can be returned immediately; if not, then it is computed and stored in the cache before being returned.

Listing 8–1. *Constants and Functions for the Lancelot Protocol*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 8 - squares.py
# Using memcached to cache expensive results.

import memcache, random, time, timeit
mc = memcache.Client(['127.0.0.1:11211'])

def compute_square(n):
»   value = mc.get('sq:%d' % n)
»   if value is None:
```

```
»   »   time.sleep(0.001)  # pretend that computing a square is expensive
»   »   value = n * n
»   »   mc.set('sq:%d' % n, value)
»   return value

def make_request():
»   compute_square(random.randint(0, 5000))

print 'Ten successive runs:',
for i in range(1, 11):
»   print '%.2fs' % timeit.timeit(make_request, number=2000),
print
```

The Memcached daemon needs to be running on your machine at port 11211 for this example to succeed. For the first few hundred requests, of course, the program will run at its usual speed. But as the cache begins to accumulate more requests, it is able to accelerate an increasingly large fraction of them.

After a few thousand requests into the domain of 5,000 possible values, the program is showing a substantial speed-up, and runs five times faster on its tenth run of 2,000 requests than on its first:

```
$ python squares.py
Ten successive runs: 2.75s 1.98s 1.51s 1.14s 0.90s 0.82s 0.71s 0.65s 0.58s 0.55s
```

This pattern is generally characteristic of caching: a gradual improvement as the cache begins to cover the problem domain, and then stability as either the cache fills or the input domain has been fully covered.

In a real application, what kind of data might you want to write to the cache?

Many programmers simply cache the lowest level of expensive call, like queries to a database, filesystem, or external service. It can, after all, be easy to understand which items can be cached for how long without making information too out-of-date; and if a database row changes, then perhaps the cache can even be preemptively cleared of stale items related to the changed value. But sometimes there can be great value in caching intermediate results at higher levels of the application, like data structures, snippets of HTML, or even entire web pages. That way, a cache hit prevents not only a database access but also the cost of turning the result into a data structure and then into rendered HTML.

There are many good introductions and in-depth guides that are linked to from the Memcached site, as well as a surprisingly extensive FAQ, as though the Memcached developers have discovered that catechism is the best way to teach people about their service. I will just make some general points here.

First, keys have to be unique, so developers tend to use prefixes and encodings to keep distinct the various classes of objects they are storing—you often see things like `user:19`, `mypage:/node/14`, or even the entire text of a SQL query used as a key. Keys can be only 250 characters long, but by using a strong hash function, you might get away with lookups that support longer strings. The values stored in Memcached, by the way, can be at most 1MB in length.

Second, you must always remember that Memcached is a cache; it is ephemeral, it uses RAM for storage, and, if re-started, it remembers nothing that you have ever stored! Your application should always be able to recover if the cache should disappear.

Third, make sure that your cache does not return data that is too old to be accurately presented to your users. "Too old" depends entirely upon your problem domain; a bank balance probably needs to be absolutely up-to-date, while "today's top headline" can probably be an hour old. There are three approaches to solving this problem:

- Memcached will let you set an expiration date and time on each item that you place in the cache, and it will take care of dropping these items silently when the time comes.

- You can reach in and actively invalidate particular cache entries at the moment they become no longer valid.

- You can rewrite and replace entries that are invalid instead of simply removing them, which works well for entries that might be hit dozens of times per second: instead of all of those clients finding the missing entry and all trying to simultaneously recompute it, they find the rewritten entry there instead. For the same reason, pre-populating the cache when an application first comes up can also be a crucial survival skill for large sites.

As you might guess, decorators are a very popular way to add caching in Python since they wrap function calls without changing their names or signatures. If you look at the Python Package Index, you will find several decorator cache libraries that can take advantage of Memcached, and two that target popular web frameworks: `django-cache-utils` and the `plone.memoize` extension to the popular CMS.

Finally, as always when persisting data structures with Python, you will have to either create a string representation yourself (unless, of course, the data you are trying to store is itself simply a string!), or use a module like `pickle` or `json`. Since the point of Memcached is to be fast, and you will be using it at crucial points of performance, I recommend doing some quick tests to choose a data representation that is both rich enough and also among your fastest choices. Something ugly, fast, and Python-specific like `cPickle` will probably do very well.

Memcached and Sharding

The design of Memcached illustrates an important principle that is used in several other kinds of databases, and which you might want to employ in architectures of your own: the clients *shard* the database by hashing the keys' string values and letting the hash determine which member of the cluster is consulted for each key.

To understand why this is effective, consider a particular key/value pair—like the key `sq:42` and the value `1764` that might be stored by Listing 8–1. To make the best use of the RAM it has available, the Memcached cluster wants to store this key and value exactly once. But to make the service fast, it wants to avoid duplication without requiring any coordination between the different servers or communication between all of the clients.

This means that all of the clients, without any other information to go on than (a) the key and (b) the list of Memcached servers with which they are configured, need some scheme for working out where that piece of information belongs. If they fail to make the same decision, then not only might the key and value be copied on to several servers and reduce the overall memory available, but also a client's attempt to remove an invalid entry could leave other invalid copies elsewhere.

The solution is that the clients all implement a single, stable algorithm that can turn a key into an integer *n* that selects one of the servers from their list. They do this by using a "hash" algorithm, which mixes the bits of a string when forming a number so that any pattern in the string is, hopefully, obliterated.

To see why patterns in key values must be obliterated, consider Listing 8–2. It loads a dictionary of English words (you might have to download a dictionary of your own or adjust the path to make the script run on your own machine), and explores how those words would be distributed across four servers if they were used as keys. The first algorithm tries to divide the alphabet into four roughly equal sections and distributes the keys using their first letter; the other two algorithms use hash functions.

Listing 8–2. *Two Schemes for Assigning Data to Servers*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 8 - hashing.py
# Hashes are a great way to divide work.

import hashlib
```

```
def alpha_shard(word):
    """Do a poor job of assigning data to servers by using first letters."""
    if word[0] in 'abcdef':
        return 'server0'
    elif word[0] in 'ghijklm':
        return 'server1'
    elif word[0] in 'nopqrs':
        return 'server2'
    else:
        return 'server3'

def hash_shard(word):
    """Do a great job of assigning data to servers using a hash value."""
    return 'server%d' % (hash(word) % 4)

def md5_shard(word):
    """Do a great job of assigning data to servers using a hash value."""
    # digest() is a byte string, so we ord() its last character
    return 'server%d' % (ord(hashlib.md5(word).digest()[-1]) % 4)

words = open('/usr/share/dict/words').read().split()

for function in alpha_shard, hash_shard, md5_shard:
    d = {'server0': 0, 'server1': 0, 'server2': 0, 'server3': 0}
    for word in words:
        d[function(word.lower())] += 1
    print function.__name__[:-6], d
```

The `hash()` function is Python's own built-in hash routine, which is designed to be blazingly fast because it is used internally to implement Python dictionary lookup. The MD5 algorithm is much more sophisticated because it was actually designed as a cryptographic hash; although it is now considered too weak for security use, using it to distribute load across servers is fine (though slow).

The results show quite plainly the danger of trying to distribute load using any method that could directly expose the patterns in your data:

```
$ python hashing.py
alpha {'server0': 35203, 'server1': 22816, 'server2': 28615, 'server3': 11934}
hash {'server0': 24739, 'server1': 24622, 'server2': 24577, 'server3': 24630}
md5 {'server0': 24671, 'server1': 24726, 'server2': 24536, 'server3': 24635}
```

You can see that distributing load by first letters results in server 0 getting more than three times the load of server 3, even though it was assigned only six letters instead of seven! The hash routines, however, both performed like champions: despite all of the strong patterns that characterize not only the first letters but also the entire structure and endings of English words, the hash functions scattered the words very evenly across the four buckets.

Though many data sets are not as skewed as the letter distributions of English words, sharded databases like Memcached always have to contend with the appearance of patterns in their input data.

Listing 8–1, for example, was not unusual in its use of keys that always began with a common prefix (and that were followed by characters from a very restricted alphabet: the decimal digits). These kinds of obvious patterns are why sharding should always be performed through a hash function.

Of course, this is an implementation detail that you can often ignore when you use a database system like Memcached that supports sharding internally. But if you ever need to design a service of your own that automatically assigns work or data to nodes in a cluster in a way that needs to be reproducible, then you will find the same technique useful in your own code.

Message Queues

Message queue protocols let you send reliable chunks of data called (predictably) *messages*. Typically, a queue promises to transmit messages reliably, and to deliver them atomically: a message either arrives whole and intact, or it does not arrive at all. Clients never have to loop and keep calling something like `recv()` until a whole message has arrived.

The other innovation that message queues offer is that, instead of supporting only the point-to-point connections that are possible with an IP transport like TCP, you can set up all kinds of topologies between messaging clients. Each brand of message queue typically supports several topologies.

A *pipeline* topology is the pattern that perhaps best resembles the picture you have in your head when you think of a queue: a producer creates messages and submits them to the queue, from which the messages can then be received by a consumer. For example, the front-end web machines of a photo-sharing web site might accept image uploads from end users and list the incoming files on an internal queue. A machine room full of servers could then read from the queue, each receiving one message for each read it performs, and generate thumbnails for each of the incoming images. The queue might get long during the day and then be short or empty during periods of relatively low use, but either way the front-end web servers are freed to quickly return a page to the waiting customer, telling them that their upload is complete and that their images will soon appear in their photostream.

A *publisher-subscriber* topology looks very much like a pipeline, but with a key difference. The pipeline makes sure that every queued message is delivered to exactly one consumer—since, after all, it would be wasteful for two thumbnail servers to be assigned the same photograph. But subscribers typically want to receive all of the messages that are being enqueued by each publisher—or else they want to receive every message that matches some particular topic. Either way, a publisher-subscriber model supports messages that fan out to be delivered to every interested subscriber. This kind of queue can be used to power external services that need to push events to the outside world, and also to form a fabric that a machine room full of servers can use to advertise which systems are up, which are going down for maintenance, and that can even publish the addresses of other message queues as they are created and destroyed.

Finally, a *request-reply* pattern is often the most complex because messages have to make a round-trip. Both of the previous patterns placed very little responsibility on the producer of a message: they connect to the queue, transmit their message, and are done. But a message queue client that makes a request has to stay connected and wait for the corresponding reply to be delivered back to it. The queue itself, to support this, has to feature some sort of addressing scheme by which replies can be directed to the correct client that is still sitting and waiting for it. But for all of its underlying complexity, this is probably the most powerful pattern of all, since it allows the load of dozens or hundreds of clients to be spread across equally large numbers of servers without any effort beyond setting up the message queue. And since a good message queue will allow servers to attach and detach without losing messages, this topology allows servers to be brought down for maintenance in a way that is invisible to the population of client machines.

Request-reply queues are a great way to connect lightweight workers that can run together by the hundreds on a particular machine—like, say, the threads of a web server front end—to database clients or file servers that sometimes need to be called in to do heavier work on the front end's behalf. And the request-reply pattern is a natural fit for RPC mechanisms, with an added benefit not usually offered by simpler RPC systems: that many consumers or many producers can all be attached to the same queue in a fan-in or fan-out work pattern, without either group of clients knowing the difference.

Using Message Queues from Python

Messaging seems to have been popular in the Java world before it started becoming the rage among Python programmers, and the Java approach was interesting: instead of defining a protocol, their community defined an API standard called the JMS on which the various message queue vendors could standardize. This gave them each the freedom—but also the responsibility—to invent and adopt some particular on-the-wire protocol for their particular message queue, and then hide it behind their own implementation of the standard API. Their situation, therefore, strongly resembles that of SQL databases under Python today: databases all use different on-the-wire protocols, and no one can really do anything to improve that situation. But you can at least write your code against the DB-API 2.0 (PEP 249) and hopefully run against several different database libraries as the need arises.

A competing approach that is much more in line with the Internet philosophy of open standards, and of competing client and server implementations that can all interoperate, is the Advanced Message Queuing Protocol (AMQP), which is gaining significant popularity among Python programmers. A favorite combination at the moment seems to be the RabbitMQ message broker, written in Erlang, with a Python AMQP client library like Carrot.

There are several AMQP implementations currently listed in the Python Package Index, and their popularity will doubtless wax and wane over the years that this book remains relevant. Future readers will want to read recent blog posts and success stories to learn about which libraries are working out best, and check for which packages have been released recently and are showing active development. Finally, you might find that a particular implementation is a favorite in combination with some other technology you are using—as Celery currently seems a favorite with Django developers—and that might serve as a good guide to choosing a library.

An alternative to using AMQP and having to run a central broker, like RabbitMQ or Apache Qpid, is to use ØMQ, the "Zero Message Queue," which was invented by the same company as AMQP but moves the messaging intelligence from a centralized broker into every one of your message client programs. The ØMQ library embedded in each of your programs, in other words, lets your code spontaneously build a messaging fabric without the need for a centralized broker. This involves several differences in approach from an architecture based on a central broker that can provide reliability, redundancy, retransmission, and even persistence to disk. A good summary of the advantages and disadvantages is provided at the ØMQ web site: `www.zeromq.org/docs:welcome-from-amqp`.

How should you approach this range of possible solutions, or evaluate other message queue technologies or libraries that you might find mentioned on Python blogs or PyCon talks?

You should probably focus on the particular message pattern that you need to implement. If you are using messages as simply a lightweight and load-balanced form of RPC behind your front-end web machines, for example, then ØMQ might be a great choice; if a server reboots and its messages are lost, then either users will time out and hit reload, or you can teach your front-end machines to resubmit their requests after a modest delay. But if your messages each represent an unrepeatable investment of effort by one of your users—if, for example, your social network site saves user status updates by placing them on a queue and then telling the users that their update succeeded—then a message broker with strong guarantees against message loss will be the only protection your users will have against having to re-type the same status later when they notice that it never got posted.

Listing 8–3 shows some of the patterns that can be supported when message queues are used to connect different parts of an application. It requires ØMQ, which you can most easily make available to Python by creating a virtual environment and then typing the following:

```
$ pip install pyzmq-static
```

The listing uses Python threads to create a small cluster of six different services. One pushes a constant stream of words on to a pipeline. Three others sit ready to receive a word from the pipeline; each word wakes one of them up. The final two are request-reply servers, which resemble remote procedure endpoints (see Chapter 18) and send back a message for each message they receive.

Listing 8–3. *Two Schemes for Assigning Data to Servers*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 8 - queuecrazy.py
# Small application that uses several different message queues

import random, threading, time, zmq
zcontext = zmq.Context()

def fountain(url):
    """Produces a steady stream of words."""
    zsock = zcontext.socket(zmq.PUSH)
    zsock.bind(url)
    words = [ w for w in dir(__builtins__) if w.islower() ]
    while True:
        zsock.send(random.choice(words))
        time.sleep(0.4)

def responder(url, function):
    """Performs a string operation on each word received."""
    zsock = zcontext.socket(zmq.REP)
    zsock.bind(url)
    while True:
        word = zsock.recv()
        zsock.send(function(word))  # send the modified word back

def processor(n, fountain_url, responder_urls):
    """Read words as they are produced; get them processed; print them."""
    zpullsock = zcontext.socket(zmq.PULL)
    zpullsock.connect(fountain_url)

    zreqsock = zcontext.socket(zmq.REQ)
    for url in responder_urls:
        zreqsock.connect(url)

    while True:
        word = zpullsock.recv()
        zreqsock.send(word)
        print n, zreqsock.recv()

def start_thread(function, *args):
    thread = threading.Thread(target=function, args=args)
    thread.daemon = True  # so you can easily Control-C the whole program
    thread.start()

start_thread(fountain, 'tcp://127.0.0.1:6700')
start_thread(responder, 'tcp://127.0.0.1:6701', str.upper)
start_thread(responder, 'tcp://127.0.0.1:6702', str.lower)
for n in range(3):
    start_thread(processor, n + 1, 'tcp://127.0.0.1:6700',
                    ['tcp://127.0.0.1:6701', 'tcp://127.0.0.1:6702'])
time.sleep(30)
```

The two request-reply servers are different—one turns each word it receives to uppercase, while the other makes its words all lowercase—and you can tell the three processors apart by the fact that each is assigned a different integer. The output of the script shows you how the words, which originate from a single source, get evenly distributed among the three workers, and by paying attention to the capitalization, you can see that the three workers are spreading their requests among the two request-reply servers:

```
1 HASATTR
2 filter
3 reduce
1 float
2 BYTEARRAY
3 FROZENSET
```

In practice, of course, you would usually use message queues for connecting entirely different servers in a cluster, but even these simple threads should give you a good idea of how a group of services can be arranged.

How Message Queues Change Programming

Whatever message queue you use, I should warn you that it may very well cause a revolution in your thinking and eventually make large changes to the very way that you construct large applications.

Before you encounter message queues, you tend to consider the function or method call to be the basic mechanism of cooperation between the various pieces of your application. And so the problem of building a program, up at the highest level, is the problem of designing and writing all of its different pieces, and then of figuring out how they will find and invoke one another. If you happen to create multiple threads or processes in your application, then they tend to correspond to outside demands—like having one server thread per external client—and to execute code from across your entire code base in the performance of your duties. The thread might receive a submitted photograph, then call the routine that saves it to storage, then jump into the code that parses and saves the photograph's metadata, and then finally execute the image processing code that generates several thumbnails. This single thread of control may wind up touching every part of your application, and so the task of scaling your service becomes that of duplicating this one piece of software over and over again until you can handle your client load.

If the best tools available for some of your sub-tasks happen to be written in other languages—if, for example, the thumbnails can best be processed by some particular library written in the C language—then the seams or boundaries between different languages take the form of Python extension libraries or interfaces like `ctypes` that can make the jump between different language runtimes.

Once you start using message queues, however, your entire approach toward service architecture may begin to experience a Copernican revolution.

Instead of thinking of complicated extension libraries as the natural way for different languages to interoperate, you will not be able to help but notice that your message broker of choice supports many different language bindings. Why should a single thread of control on one processor, after all, have to wind its way through a web framework, then a database client, and then an imaging library, when you could make each of these components a separate client of the messaging broker and connect the pieces with language-neutral messages?

You will suddenly realize not only that a dedicated thumbnail service might be quite easy to test and debug, but also that running it as a separate service means that it can be upgraded and expanded without any disruption to your front-end web servers. New servers can attach to the message queue, old ones can be decommissioned, and software updates can be pushed out slowly to one back end after another without the front-end clients caring at all. The queued message, rather than the library API, will become the fundamental point of rendezvous in your application.

And all of this can have a startling impact on your approach toward concurrency, especially where shared resources are concerned.

When all of your application's work and resources are present within a single address space containing dozens of Python packages and libraries, then it can seem like semaphores, locks, and shared data structures—despite all of the problems inherent in using them correctly—are the natural mechanisms for cooperation.

But message services offer a different model: that of small, autonomous services attached to a common queue, that let the queue take care of getting information—namely, messages—safely back and forth between dozens of different processes. Suddenly, you will find yourself writing Python components that begin to take on the pleasant concurrent semantics of Erlang function calls: they will accept a request, use their carefully husbanded resources to generate a response, and never once explicitly touch a shared data structure. The message queue will not only take care of shuttling data back and forth, but by letting client procedures that have sent requests wait on server procedures that are generating results, the message queue also provides a well-defined synchrony with which your processes can coordinate their activity.

If you are not yet ready to try external message queues, be sure to at least look very closely at the Python Standard Library when writing concurrent programs, paying close attention to the `queue` module and also to the between-process `Queue` that is offered by the `multiprocessing` library. Within the confines of a single machine, these mechanisms can get you started on writing application components as scalable producers and consumers.

Finally, if you are writing a large application that is sending huge amounts of data in one direction using the pipeline pattern, then you might also want to check out this resource: `http://wiki.python.org/moin/FlowBasedProgramming`.

It will point you toward resources related to Python and "flow-based" programming, which steps back from the idea of messages to the more general idea of information flowing downstream from an origin, through various processing steps, and finally to a destination that saves or displays the result. This can be a very natural way to express various scientific computations, as well as massively data-driven tasks like searching web server log files for various patterns. Some flow-based systems even support the use of a graphical interface, which can let scientists and other researchers who might be unfamiliar with programming build quite sophisticated data processing stacks.

One final note: do not let the recent popularity of message queues mislead you into thinking that the messaging pattern itself is a recent phenomenon! It is not. Message queues are merely the formalization of an ages-old architecture that would originally have involved piles of punch cards waiting for processing, and that in more recent incarnations included things like "incoming" FTP folders full of files that were submitted for processing. The modern libraries are simply a useful and general implementation of a very old wheel that has been re-invented countless times.

Map-Reduce

Traditionally, if you wanted to distribute a large task across several racks of machine-room servers, then you faced two quite different problems. First, of course, you had to write code that could be assigned a small part of the problem and solve it, and then write code that could assemble the various answers from each node back into one big answer to the original question.

But, finally, you would also have wound up writing a lot of code that had little to do with your problem at all: the scripts that would push your code out to all of the servers in the cluster, then run it, and then finally collect the data back together using the network or a shared file system.

The idea of a map-reduce system is to eliminate that last step in distributing a large computation, and to offer a framework that will distribute data and execute code without your having to worry about the underlying distribution mechanisms. Most frameworks also implement precautions that are often not present in homemade parallel computations, like the ability to seamlessly re-submit tasks to other nodes if some of the cluster servers fail during a particular computation. In fact, some map-reduce frameworks will happily let you unplug and reboot machines for routine maintenance even while the

cluster is busy with a computation, and will quietly work around the unavailable nodes without disturbing the actual application in the least.

Note that there are two quite different reasons for distributing a computation. One kind of task simply requires a lot of CPU. In this case, the cluster nodes do not start off holding any data relevant to the problem; they have to be loaded with both their data set and code to run against it. But another kind of task involves a large data set that is kept permanently distributed across the nodes, making them asymmetric workers who are each, so to speak, the expert on some particular slice of the data. This approach could be used, for example, by an organization that has saved years of web logs across dozens of machines, and wants to perform queries where each machine in the cluster computes some particular tally, or looks for some particular pattern, in the few months of data for which it is uniquely responsible.

Although a map-reduce framework might superficially resemble the Beowulf clusters pioneered at NASA in the 1990s, it imposes a far more specific semantics on the phases of computation than did the generic message-passing libraries that tended to power Beowulf's. Instead, a map-reduce framework takes responsibility for both distributing tasks and assembling an answer, by imposing structure on the processing code submitted by programmers:

- The task under consideration needs to be broken into two pieces, one called the *map* operation, and the other *reduce*.
- The two operations bear some resemblance to the Python built-in functions of that name (which Python itself borrowed from the world of functional programming); imagine how one might split across several servers the tasks of summing the squares of many integers:

```
>>> squares = map(lambda n: n*n, range(11))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> import operator
>>> reduce(operator.add, squares)
385
```

- The mapping operation should be prepared to run once on some particular slice of the overall problem or data set, and to produce a tally, table, or response that summarizes its findings for that slice of the input.
- The reduce operation is then exposed to the outputs of the mapping functions, to combine them together into an ever-accumulating answer. To use the map-reduce cluster's power effectively, frameworks are not content to simply run the reduce function on one node once all of the dozens or hundreds of active machines have finished the mapping stage. Instead, the reduce function is run in parallel on many nodes at once, each considering the output of a handful of map operations, and then these intermediate results are combined again and again in a tree of computations until a final reduce step produces output for the whole input.
- Thus, map-reduce frameworks require the programmer to be careful, and write reduce functions that can be safely run on the same data over and over again; but the specific guidelines and guarantees with respect to reduce can vary, so check the tutorials and user guides to specific map-reduce frameworks that interest you.

Many map-reduce implementations are commercial and cloud-based, because many people need them only occasionally, and paying to run their operation on Google MapReduce or Amazon Elastic MapReduce is much cheaper than owning enough servers themselves to set up Hadoop or some other self-hosted solution.

Significantly, the programming APIs for the various map-reduce solutions are often similar enough that Python interfaces can simply paper over the differences and offer the same interface regardless of

which back end you are using; for example, the `mrjob` library supports both Hadoop and Amazon. Some programmers avoid using a specific API altogether, and submit their Python programs to Hadoop as external scripts that it should run using its "streaming" module that uses the standard input and output of a subprocess to communicate—the CGI-BIN of the map-reduce world, I suppose.

Note that some of the new generation of NoSQL databases, like CouchDB and MongoDB, offer the map-reduce pattern as a way to run distributed computations across your database, or even—in the case of CouchDB—as the usual way to create indexes. Conversely, each map-reduce framework tends to come with its own brand of distributed filesystem or file-like storage that is designed to be efficiently shared across many nodes.

Summary

Serving thousands or millions of customers has become a routine assignment for application developers in the modern world, and several key technologies have emerged to help them meet this scale—and all of them can easily be accessed from Python.

The most popular may be Memcached, which combines the free RAM across all of the servers on which it is installed into a single large LRU cache. As long as you have some procedure for invalidating or replacing entries that become out of date—or an interface with components that are allowed to go seconds, minutes, or hours out of date before needing to be updated—Memcached can remove massive load from your database or other back-end storage. It can also be inserted at several different points in your processing; instead of saving the result of an expensive database query, for example, it might be even better to simply cache the web widget that ultimately gets rendered. You can assign an expiration data to cache entries as well, in which case Memcached will remove them for you when they have grown too old.

Message queues provide a point of coordination and integration for different parts of your application that may require different hardware, load balancing techniques, platforms, or even programming languages. They can take responsibility for distributing messages among many waiting consumers or servers in a way that is not possible with the single point-to-point links offered by normal TCP sockets, and can also use a database or other persistent storage to assure that updates to your service are not lost if the server goes down. Message queues also offer resilience and flexibility, since if some part of your system temporarily becomes a bottleneck, then the message queue can absorb the shock by allowing many messages to queue up for that service. By hiding the population of servers or processes that serve a particular kind of request, the message queue pattern also makes it easy to disconnect, upgrade, reboot, and reconnect servers without the rest of your infrastructure noticing.

Finally, the map-reduce pattern provides a cloud-style framework for distributed computation across many processors and, potentially, across many parts of a large data set. Commercial offerings are available from companies like Google and Amazon, while the Hadoop project is the foremost open source alternative—but one that requires users to build server farms of their own, instead of renting capacity from a cloud service.

If any of these patterns sound like they address a problem of yours, then search the Python Package Index for good leads on Python libraries that might implement them. The state of the art in the Python community can also be explored through blogs, tweets, and especially Stack Overflow, since there is a strong culture there of keeping answers up-to-date as solutions age and new ones emerge.

CHAPTER 9

HTTP

The protocols of yore tended to be dense, binary, and decipherable only by Boolean machine logic. But the workhorse protocol of the World Wide Web, named the Hypertext Transfer Protocol (HTTP), is instead based on friendly, mostly-human-readable text. There is probably no better way to start this chapter than to show you what an actual request and response looks like; that way, you will already know the layout of a whole request as we start digging into each of its features.

Consider what happens when you ask the `urllib2` Python Standard Library to open this URL, which is the RFC that defines the HTTP protocol itself: `www.ietf.org/rfc/rfc2616.txt`

The library will connect to the IETF web site, and send it an HTTP request that looks like this:

```
GET /rfc/rfc2616.txt HTTP/1.1
Accept-Encoding: identity
Host: www.ietf.org
Connection: close
User-Agent: Python-urllib/2.6
```

As you can see, the format of this request is very much like that of the headers of an e-mail message—in fact, both HTTP and e-mail messages define their header layout using the same standard: RFC 822. The HTTP response that comes back over the socket also starts with a set of headers, but then also includes a *body* that contains the document itself that has been requested (which I have truncated):

```
HTTP/1.1 200 OK
Date: Wed, 27 Oct 2010 17:12:01 GMT
Server: Apache/2.2.4 (Linux/SUSE) mod_ssl/2.2.4 OpenSSL/0.9.8e PHP/5.2.6 with Suhosin-➥
Patch mod_python/3.3.1 Python/2.5.1 mod_perl/2.0.3 Perl/v5.8.8
Last-Modified: Fri, 11 Jun 1999 18:46:53 GMT
ETag: "1cad180-67187-31a3e140"
Accept-Ranges: bytes
Content-Length: 422279
Vary: Accept-Encoding
Connection: close
Content-Type: text/plain

Network Working Group                                      R. Fielding
Request for Comments: 2616                                   UC Irvine
Obsoletes: 2068                                              J. Gettys
Category: Standards Track                                   Compaq/W3C
...
```

Note that those last four lines are the beginning of RFC 2616 itself, not part of the HTTP protocol.

Two of the most important features of this format are not actually visible here, because they pertain to whitespace. First, every header line is concluded by a two-byte carriage-return linefeed sequence, or `'\r\n'` in Python. Second, both sets of headers are terminated—in HTTP, headers are *always*

terminated—by a blank line. You can see the blank line between the HTTP response and the document that follows, of course; but in this book, the blank line that follows the HTTP request headers is probably invisible. When viewed as raw characters, the headers end where two end-of-line sequences follow one another with nothing in between them:

```
…Penultimate-Header: value\r\nLast-Header: value\r\n\r\n
```

Everything after that final `\n` is data that belongs to the document being returned, and not to the headers. It is very important to get this boundary strictly correct when writing an HTTP implementation because, although text documents might still be legible if some extra whitespace works its way in, images and other binary data would be rendered unusable.

As this chapter proceeds to explore the features of HTTP, we are going to illustrate the protocol using several modules that come built-in to the Python Standard Library, most notably its `urllib2` module. Some people advocate the use of HTTP libraries that require less fiddling to behave like a normal browser, like `mechanize` or even PycURL, which you can find at these locations:

```
http://wwwsearch.sourceforge.net/mechanize/
http://pycurl.sourceforge.net/
```

But `urllib2` is powerful and, when understood, convenient enough to use that I am going to support the Python "batteries included" philosophy and feature it here. Plus, it supports a pluggable system of request handlers that we will find very useful as we progress from simple to complex HTTP exchanges in the course of the chapter.

If you examine the source code of `mechanize`, you will find that it actually builds on top of `urllib2`; thus, it can be an excellent source of hints and patterns for adding features to the classes already in the Standard Library. It even supports cookies out of the box, which `urllib2` makes you enable manually. Note that some features, like gzip compression, are not available by default in either framework, although `mechanize` makes compression much easier to turn on.

I must acknowledge that I have myself learned `urllib2,` not only from its documentation, but from the web site of Michael Foord and from the *Dive Into Python* book by Mark Pilgrim. Here are links to each of those resources:

```
http://www.voidspace.org.uk/python/articles/urllib2.shtml
http://diveintopython.org/toc/index.html
```

And, of course, RFC 2616 (the link was given a few paragraphs ago) is the best place to start if you are in doubt about some technical aspect of the protocol itself.

URL Anatomy

Before tackling the inner workings of HTTP, we should pause to settle a bit of terminology surrounding Uniform Resource Locators (URLs), the wonderful strings that tell your web browser how to fetch resources from the World Wide Web. They are a subclass of the full set of possible Uniform Resource Identifiers (URIs); specifically, they are URIs constructed so that they give instructions for fetching a document, instead of serving only as an identifier.

For example, consider a very simple URL like the following: `http://python.org`

If submitted to a web browser, this URL is interpreted as an order to resolve the host name `python.org` to an IP address (see Chapter 4), make a TCP connection to that IP address at the standard HTTP port 80 (see Chapter 3), and then ask for the root document `/` that lives at that site.

Of course, many URLs are more complicated. Imagine, for example, that there existed a service offering pre-scaled thumbnail versions of various corporate logos for an international commerce site we were writing. And imagine that we wanted the logo for Nord/LB, a large German bank. The resulting URL might look something like this: `http://example.com:8080/Nord%2FLB/logo?shape=square&dpi=96`

Here, the URL specifies more information than our previous example did:

- The protocol will, again, be HTTP.
- The hostname `example.com` will be resolved to an IP.
- This time, port 8080 will be used instead of 80.
- Once a connection is complete, the remote server will be asked for the resource named:

```
/Nord%2FLB/logo?shape=square&dpi=96
```

Web servers, in practice, have absolute freedom to interpret URLs as they please; however, the intention of the standard is that this URL be parsed into two question-mark-delimited pieces. The first is a path consisting of two elements:

- A `Nord/LB` path element.
- A `logo` path element.

The string following the `?` is interpreted as a query containing two terms:

- A `shape` parameter whose value is `square`.
- A `dpi` parameter whose value is `96`.

Thus can complicated URLs be built from simple pieces.

Any characters beyond the alphanumerics, a few punctuation marks—specifically the set `$-_.+!*'(),`—and the special delimiter characters themselves (like the slashes) must be *percent-encoded* by following a percent sign `%` with the two-digit hexadecimal code for the character. You have probably seen `%20` used for a space in a URL, for example, and `%2F` when a slash needs to appear.

The case of `%2F` is important enough that we ought to pause and consider that last URL again. Please note that the following URL paths are *not* equivalent:

```
Nord%2FLB%2Flogo
Nord%2FLB/logo
Nord/LB/logo
```

These are *not* three versions of the same URL path! Instead, their respective meanings are as follows:

- A single path component, named `Nord/LB/logo`.
- Two path components, `Nord/LB` and `logo`.
- Three separate path components `Nord`, `LB`, and `logo`.

These distinctions are especially crucial when web clients parse relative URLs, which we will discuss in the next section.

The most important Python routines for working with URLs live, appropriately enough, in their own module:

```
>>> from urlparse import urlparse, urldefrag, parse_qs, parse_qsl
```

At least, the functions live together in recent versions of Python—for versions of Pythons older than 2.6, two of them live in the `cgi` module instead:

```
# For Python 2.5 and earlier
>>> from urlparse import urlparse, urldefrag
>>> from cgi import parse_qs, parse_qsl
```

With these routines, you can get large and complex URLs like the example given earlier and turn them into their component parts, with RFC-compliant parsing already implemented for you:

```
>>> p = urlparse('http://example.com:8080/Nord%2FLB/logo?shape=square&dpi=96')
>>> p
ParseResult(scheme='http', netloc='example.com:8080', path='/Nord%2FLB/logo',
» » »   params='', query='shape=square&dpi=96', fragment='')
```

The query string that is offered by the ParseResult can then be submitted to one of the parsing routines if you want to interpret it as a series of key-value pairs, which is a standard way for web forms to submit them:

```
>>> parse_qs(p.query)
{'shape': ['square'], 'dpi': ['96']}
```

Note that each value in this dictionary is a list, rather than simply a string. This is to support the fact that a given parameter might be specified several times in a single URL; in such cases, the values are simply appended to the list:

```
>>> parse_qs('mode=topographic&pin=Boston&pin=San%20Francisco')
{'mode': ['topographic'], 'pin': ['Boston', 'San Francisco']}
```

This, you will note, preserves the order in which values arrive; of course, this does not preserve the order of the parameters themselves because dictionary keys do not remember any particular order. If the order is important to you, then use the `parse_qsl()` function instead (the `l` must stand for "list"):

```
>>> parse_qsl('mode=topographic&pin=Boston&pin=San%20Francisco')
[('mode', 'topographic'), ('pin', 'Boston'), ('pin', 'San Francisco')]
```

Finally, note that an "anchor" appended to a URL after a # character is *not* relevant to the HTTP protocol. This is because any anchor is stripped off and is not turned into part of the HTTP request. Instead, the anchor tells a web client to jump to some particular section of a document *after* the HTTP transaction is complete and the document has been downloaded. To remove the anchor, use `urldefrag()`:

```
>>> u = 'http://docs.python.org/library/urlparse.html#urlparse.urldefrag'
>>> urldefrag(u)
('http://docs.python.org/library/urlparse.html', 'urlparse.urldefrag')
```

You can turn a `ParseResult` back into a URL by calling its `geturl()` method. When combined with the `urlencode()` function, which knows how to build query strings, this can be used to construct new URLs:

```
>>> import urllib, urlparse
>>> query = urllib.urlencode({'company': 'Nord/LB', 'report': 'sales'})
>>> p = urlparse.ParseResult(
...     'https', 'example.com', 'data', None, query, None)
>>> p.geturl()
'https://example.com/data?report=sales&company=Nord%2FLB'
```

Note that `geturl()` correctly escapes all special characters in the resulting URL, which is a strong argument for using this means of building URLs rather than trying to assemble strings correctly by hand.

Relative URLs

Very often, the links used in web pages do not specify full URLs, but *relative URLs* that are missing several of the usual components. When one of these links needs to be resolved, the client needs to fill in the missing information with the corresponding fields from the URL used to fetch the page in the first place.

Relative URLs are convenient for web page designers, not only because they are shorter and thus easier to type, but because if an entire sub-tree of a web site is moved somewhere else, then the links will keep working. The simplest relative links are the names of pages one level deeper than the base page:

```
>>> urlparse.urljoin('http://www.python.org/psf/', 'grants')
'http://www.python.org/psf/grants'
>>> urlparse.urljoin('http://www.python.org/psf/', 'mission')
'http://www.python.org/psf/mission'
```

Note the crucial importance of the trailing slash in the URLs we just gave to the `urljoin()` function! Without the trailing slash, the call function will decide that the *current directory* (called officially the *base URL*) is `/` rather than `/psf/`; therefore, it will replace the `psf` component entirely:

```
>>> urlparse.urljoin('http://www.python.org/psf', 'grants')
'http://www.python.org/grants'
```

Like file system paths on the POSIX and Windows operating systems, `.` can be used for the current directory and `..` is the name of the parent:

```
>>> urlparse.urljoin('http://www.python.org/psf/', './mission')
'http://www.python.org/psf/mission'
>>> urlparse.urljoin('http://www.python.org/psf/', '../news/')
'http://www.python.org/news/'
>>> urlparse.urljoin('http://www.python.org/psf/', '/dev/')
'http://www.python.org/dev'
```

And, as illustrated in the last example, a relative URL that starts with a slash is assumed to live at the top level of the same site as the original URL.

Happily, the `urljoin()` function ignores the base URL entirely if the second argument also happens to be an absolute URL. This means that you can simply pass every URL on a given web page to the `urljoin()` function, and any relative links will be converted; at the same time, absolute links will be passed through untouched:

```
# Absolute links are safe from change
>>> urlparse.urljoin('http://www.python.org/psf/', 'http://yelp.com/')
'http://yelp.com/'
```

As we will see in the next chapter, converting relative to absolute URLs is important whenever we are packaging content that lives under one URL so that it can be displayed at a different URL.

Instrumenting urllib2

We now turn to the HTTP protocol itself. Although its on-the-wire appearance is usually an internal detail handled by web browsers and libraries like `urllib2`, we are going to adjust its behavior so that we can see the protocol printed to the screen. Take a look at Listing 9–1.

***Listing 9–1.** An HTTP Request and Response that Prints All Headers*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 9 - verbose_handler.py
# HTTP request handler for urllib2 that prints requests and responses.

import StringIO, httplib, urllib2

class VerboseHTTPResponse(httplib.HTTPResponse):
    def _read_status(self):
        s = self.fp.read()
        print '-' * 20, 'Response', '-' * 20
        print s.split('\r\n\r\n')[0]
        self.fp = StringIO.StringIO(s)
        return httplib.HTTPResponse._read_status(self)

class VerboseHTTPConnection(httplib.HTTPConnection):
    response_class = VerboseHTTPResponse
    def send(self, s):
        print '-' * 50
        print s.strip()
        httplib.HTTPConnection.send(self, s)

class VerboseHTTPHandler(urllib2.HTTPHandler):
    def http_open(self, req):
        return self.do_open(VerboseHTTPConnection, req)
```

To allow for customization, the `urllib2` library lets you bypass its vanilla `urlopen()` function and instead build an *opener* full of handler classes of your own devising—a fact that we will use repeatedly as this chapter progresses. Listing 9–1 provides exactly such a handler class by performing a slight customization on the normal HTTP handler. This customization prints out both the outgoing request and the incoming response instead of keeping them both hidden.

For many of the following examples, we will use an `opener` object that we build right here, using the handler from Listing 9–1:

```
>>> from verbose_http import VerboseHTTPHandler
>>> import urllib, urllib2
>>> opener = urllib2.build_opener(VerboseHTTPHandler)
```

You can try using this opener against the URL of the RFC that we mentioned at the beginning of this chapter:

```
opener.open('http://www.ietf.org/rfc/rfc2616.txt')
```

The result will be a printout of the same HTTP request and response that we used as our example at the start of the chapter. We can now use this `opener` to examine every part of the HTTP protocol in more detail.

The GET Method

When the earliest version of HTTP was first invented, it had a single power: to issue a method called `GET` that named and returned a hypertext document from a remote server. That method is still the backbone of the protocol today.

From now on, I am going to make heavy use of ellipsis (three periods in a row: `...`) to omit parts of each HTTP request and response not currently under discussion. That way, we can more easily focus on the protocol features being described.

The `GET` method, like all HTTP methods, is the first thing transmitted as part of an HTTP request, and it is immediately followed by the request headers. For simple `GET` methods, the request simply ends with the blank line that terminates the headers so the server can immediately stop reading and send a response:

```
>>> info = opener.open('http://www.ietf.org/rfc/rfc2616.txt')
--------------------------------------------------
GET /rfc/rfc2616.txt HTTP/1.1
...
Host: www.ietf.org
...
-------------------- Response --------------------
HTTP/1.1 200 OK
...
Content-Type: text/plain
```

The opener's `open()` method, like the plain `urlopen()` function at the top level of `urllib2`, returns an *information object* that lets us examine the result of the `GET` method. You can see that the HTTP request started with a *status line* containing the HTTP version, a status code, and a short message. The `info` object makes these available as object attributes; it also lets us examine the headers through a dictionary-like object:

```
>>> info.code
200
>>> info.msg
'OK'
>>> sorted(info.headers.keys())
['accept-ranges', 'connection', 'content-length', 'content-type',
 'date', 'etag', 'last-modified', 'server', 'vary']
>>> info.headers['Content-Type']
'text/plain'
```

Finally, the `info` object is also prepared to act as a file. The HTTP response status line, the headers, and the blank line that follows them have all been read from the HTTP socket, and now the actual document is waiting to be read. As is usually the case with file objects, you can either start reading the `info` object in pieces through `read(N)` or `readline()`; or you can choose to bring the entire data stream into memory as a single string:

```
>>> print info.read().strip()
Network Working Group                                      R. Fielding
Request for Comments: 2616                                   UC Irvine
Obsoletes: 2068                                              J. Gettys
Category: Standards Track                                   Compaq/W3C
...
```

These are the first lines of the longer text file that you will see if you point your web browser at the same URL.

That, then, is the essential purpose of the `GET` method: to ask an HTTP server for a particular document, so that its contents can be downloaded—and usually displayed—on the local system.

The Host Header

You will have noted that the `GET` request line includes only the path portion of the full URL: `GET /rfc/rfc2616.txt HTTP/1.1`

The other elements have, so to speak, already been consumed. The `http` scheme determined what protocol would be spoken, and the location `www.ietf.org` was used as the hostname to which a TCP connection must be made.

And in the early versions of HTTP, this was considered enough. After all, the server could tell you were speaking HTTP to it, and surely it also knew that it was the IETF web server—if there were confusion on that point, it would presumably have been the job of the IETF system administrators to sort it out!

But in a world of six billion people and four billion IP addresses, the need quickly became clear to support servers that might host dozens of web sites at the same IP. Systems administrators with, say, twenty different domains to host within a large organization were annoyed to have to set up twenty different machines—or to give twenty separate IP addresses to one single machine—simply to work around a limitation of the HTTP/1.0 protocol.

And that is why the URL location is now included in every HTTP request. For compatibility, it has not been made part of the `GET` request line itself, but has instead been stuck into the headers under the name `Host`:

```
>>> info = opener.open('http://www.google.com/')
--------------------------------------------------
GET / HTTP/1.1
...
Host: www.google.com
...
-------------------- Response --------------------
HTTP/1.1 200 OK
...
```

Depending on how they are configured, servers might return entirely different sites when confronted with two different values for `Host`; they might present slightly different versions of the same site; or they might ignore the header altogether. But semantically, two requests with different values for `Host` are asking about two entirely different URLs.

When several sites are hosted at a single IP address, those sites are each said to be served by a *virtual host*, and the whole practice is sometimes referred to as *virtual hosting*.

Codes, Errors, and Redirection

All of the HTTP responses we have seen so far specify the `HTTP/1.1` protocol version, the return code `200`, and the message `OK`. This indicates that each page was fetched successfully. But there are many more possible response codes. The full list is, of course, in RFC 2616, but here are the most basic responses (and we will discover a few others as this chapter progresses):

- `200 OK`: The request has succeeded.
- `301 Moved Permanently`: The resource that used to live at this URL has been assigned a new URL, which is specified in the `Location:` header of the HTTP response. And any bookmarks or other local copies of the link can be safely rewritten to the new URL.

- `303 See Other`: The original URL should continue to be used for this request, but on this occasion the response can be found by retrieving a different URL—the one in the response's `Location:` header. If the operation was a `POST` or `PUT` (which we will learn about later in this chapter), then a `303` means that the operation has succeeded, and that the results can be viewed by doing a `GET` at the new location.

- `304 Not Modified`: The response would normally be a `200 OK`, but the HTTP request headers indicate that the client already possesses an up-to-date copy of the resource, so its body need not be transmitted again, and this response will contain only headers. See the section on caching later in this chapter.

- `307 Temporary Redirect`: This is like a `303`, except in the case of a `POST` or `PUT`, where a `307` means that the action has *not* succeeded but needs to be retried with another `POST` or `PUT` at the URL specified in the response `Location:` header.

- `404 Not Found`: The URL does not name a valid resource.

- `500 Internal Server Error`: The web site is broken. Programmer errors, configuration problems, and unavailable resources can all cause web servers to generate this code.

- `503 Service Unavailable`: Among the several other 500-range error messages, this may be the most common. It indicates that the HTTP request cannot be fulfilled because of some temporary and transient service failure. This is the code included when Twitter displays its famous Fail Whale, for example.

Each HTTP library makes its own choices about how to handle the various status codes. If its full stack of handlers is left in place, `urllib2` will automatically follow redirections. Return codes that cannot be handled, or that indicate any kind of error, are raised as Python exceptions:

```
>>> nonexistent_url = 'http://example.com/better-living-through-http'
>>> response = opener.open(nonexistent_url)
Traceback (most recent call last):
  ...
HTTPError: HTTP Error 404: Not Found
```

But these exception objects are special: they also contain all of the usual fields and capabilities of HTTP response information objects. Remember that many web servers include a useful human-readable document when they return an error status. Such a document might include specific information about what has gone wrong. For example, many web frameworks—at least when in development mode—will return exception tracebacks along with their `500` errors when the program trying to generate the web page crashes.

By catching the `exception`, we can both see how the HTTP response looked on the wire (thanks again to the special handler that we have installed in our `opener` object), and we can assign a name to the exception to look at it more closely:

```
>>> try:
...     response = opener.open(nonexistent_url)
... except urllib2.HTTPError, e:
...     pass
--------------------------------------------------
GET /better-living-through-http HTTP/1.1
...
-------------------- Response --------------------
HTTP/1.1 404 Not Found
Date: ...
```

```
Server: Apache
Content-Length: 285
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

As you can see, this particular web site does include a human-readable document with a 404 error; the response declares it to be an HTML page that is exactly 285 octets in length. (We will learn more about content length and types later in the chapter.) Like any HTTP response object, this exception can be queried for its status code; it can also be read like a file to see the returned page:

```
>>> e.code
404
>>> e.msg
'Not Found'
>>> e.readline()
'<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">\n'
```

If you try reading the rest of the file, then deep inside of the HTML you will see the actual error message that a web browser would display for the user:

```
>>> e.read()
'...The requested URL /better-living-through-http was not found
on this server...'
```

Redirections are very common on the World Wide Web. Conscientious web site programmers, when they undertake a major redesign, will leave 301 redirects sitting at all of their old-style URLs for the sake of bookmarks, external links, and web search results that still reference them. But the volume of redirects might be even greater for the many web sites that have a preferred host name that they want displayed for users, yet also allow users to type any of several different hostnames to bring the site up.

The issue of whether a site name begins with www` looms very large in this area. Google, for example, likes those three letters to be included, so an attempt to open the Google home page with the hostname google.com will be met with a redirect to the preferred name:

```
>>> info = opener.open('http://google.com/')
--------------------------------------------------
GET / HTTP/1.1
...
Host: google.com
...
-------------------- Response --------------------
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
...
--------------------------------------------------
GET / HTTP/1.1
...
Host: www.google.com
...
-------------------- Response --------------------
HTTP/1.1 200 OK
...
```

You can see that urllib2 has followed the redirect for us, so that the response shows only the final 200 response code:

```
>>> info.code
200
```

You cannot tell by looking at the response whether a redirect occurred. You might guess that one has taken place if the requested URL does not match the path and `Host:` header in the response, but that would leave open the possibility that a poorly written server had simply returned the wrong page. The only way that `urllib2` will record redirection is if you pass in a `Request` object instead of simply submitting the URL as a string:

```
>>> request = urllib2.Request('http://www.twitter.com')
>>> info = urllib2.urlopen(request)
>>> request.redirect_dict
{'http://twitter.com/': 1}
```

Obviously, Twitter's opinion of a leading `www` is the opposite of Google's! As you can see, it is on the request—and not the response—where `urllib2` records the series of redirections. Of course, you may someday want to manage them yourself, in which case you can create an opener with your own redirection handler that always does nothing:

```
>>> class NoRedirectHandler(urllib2.HTTPRedirectHandler):
...     def http_error_302(self, req, fp, code, msg, headers):
...         return
...     http_error_301 = http_error_303 = http_error_307 = http_error_302
>>> no_redirect_opener = urllib2.build_opener(NoRedirectHandler)
>>> no_redirect_opener.open('http://www.twitter.com')
Traceback (most recent call last):
  ...
HTTPError: HTTP Error 301: Moved Permanently
```

Catching the exception enables your application to process the redirection according to its own policies. Alternatively, you could embed your application policy in the new redirection class itself, instead of having the error method simply return (as we did here).

Payloads and Persistent Connections

By default, HTTP/1.1 servers will keep a TCP connection open even after they have delivered their response. This enables you to make further requests on the same socket and avoid the expense of creating a new socket for every piece of data you might need to download. Keep in mind that downloading a modern web page can involve fetching dozens, if not hundreds, of separate pieces of content.

The `HTTPConnection` class provided by `urllib2` lets you take advantage of this feature. In fact, all requests go through one of these objects; when you use a function like `urlopen()` or use the `open()` method on an opener object, an `HTTPConnection` object is created behind the scenes, used for that one request, and then discarded. When you might make several requests to the same site, use a persistent connection instead:

```
>>> import httplib
>>> c = httplib.HTTPConnection('www.python.org')
>>> c.request('GET', '/')
>>> original_sock = c.sock
>>> content = c.getresponse().read()  # get the whole page
>>> c.request('GET', '/about/')
>>> c.sock is original_sock
True
```

You can see here that two successive requests are indeed using the same socket object.

RFC 2616 does define a header named Connection: that can be used to explicitly indicate that a request is the last one that will be made on a socket. If we insert this header manually, then we force the HTTPConnection object to create a second socket when we ask it for a second page:

```
>>> c = httplib.HTTPConnection('www.python.org')
>>> c.request('GET', '/', headers={'Connection': 'close'})
>>> original_sock = c.sock
>>> content = c.getresponse().read()
>>> c.request('GET', '/about/')
>>> c.sock is original_sock
False
```

Note that HTTPConnection does not raise an exception when one socket closes and it has to create another one; you can keep using the same object over and over again. This holds true regardless of whether the server is accepting all of the requests over a single socket, or it is sometimes hanging up and forcing HTTPConnection to reconnect.

Back in the days of HTTP 1.0 (and earlier), closing the connection was the official way to indicate that the transmission of a document was complete. The Content-Length header is so important today largely because it lets the client read several HTTP responses off the same socket without getting confused about where the next response begins. When a length cannot be provided—say, because the server is streaming data whose end it cannot predict ahead of time—then the server can opt to use *chunked encoding*, where it sends a series of smaller pieces that are each prefixed with their length. This ensures that there is still a point in the stream where the client knows that raw data will end and HTTP instructions will recommence. RFC 2616 section 3.6.1 contains the definitive description of the chunked-encoding scheme.

POST And Forms

The POST HTTP method was designed to power web forms. When forms are used with the GET method, which is indeed their default behavior, they append the form's field values to the end of the URL:
http://www.google.com/search?q=python+language

The construction of such a URL creates a new named location that can be saved; bookmarked; referenced from other web pages; and sent in e-mails, Tweets, and text messages. And for actions like searching and selecting data, these features are perfect.

But what about a login form that accepts your e-mail address and password? Not only would there be negative security implications to having these elements appended to the form URL—such as the fact that they would be displayed on the screen in the URL bar and included in your browser history—but surely it would be odd to think of your username and password as creating a new location or page on the web site in question:

```
# Bad idea
http://example.com/welcome?email=brandon@rhodesmill.org&pw=aaz9Gog3
```

Building URLs in this way would imply that a different page exists on the example.com web site for every possible password that you could try typing. This is undesirable for obvious reasons.

And so the POST method should always be used for forms that are not constructing the name of a particular page or location on a web site, but are instead performing some action on behalf of the caller. Forms in HTML can specify that they want the browser to use POST by specifying that method in their <form> element:

```
<form name="myloginform" action="/access/dummy" method="post">
E-mail: <input type="text" name="e-mail" size="20">
Password: <input type="password" name="password" size="20">
<input type="submit" name="submit" value="Login">
</form>
```

Instead of stuffing form parameters into the URL, a POST carries them in the body of the request. We can perform the same action ourselves in Python by using urlencode to format the form parameters, and then supplying them as a second parameter to any of the urllib2 methods that open a URL. Here is a simple POST to the U.S. National Weather Service that asks about the forecast for Atlanta, Georgia:

```
>>> form = urllib.urlencode({'inputstring': 'Atlanta, GA'})
>>> response = opener.open('http://forecast.weather.gov/zipcity.php', form)
--------------------------------------------------
POST /zipcity.php HTTP/1.1
...
Content-Length: 25
Host: forecast.weather.gov
Content-Type: application/x-www-form-urlencoded
...
--------------------------------------------------
inputstring=Atlanta%2C+GA
-------------------- Response --------------------
HTTP/1.1 302 Found
...
Location: http://forecast.weather.gov/MapClick.php?CityName=Atlanta&state=GA↩
&site=FFC&textField1=33.7629&textField2=-84.4226&e=1
...
--------------------------------------------------
GET /MapClick.php?CityName=Atlanta&state=GA&site=FFC&textField1=33.7629&textField2=↩
-84.4226&e=1 HTTP/1.1
...
-------------------- Response --------------------
HTTP/1.1 200 OK
...
```

Although our opener object is putting a dashed line between each HTTP request and its payload for clarity (a blank line, you will recall, is what really separates headers and payload on the wire) you are otherwise seeing a raw HTTP POST method here. Note these features of the request-responses shown in the example above:

- The request line starts with the string POST.
- Content is provided (and thus, a Content-Length header).
- The form parameters are sent as the body.
- The Content-Type for standard web forms is x-www-form-urlencoded.

The most important thing to grasp is that GET and POST are most emphatically *not* simply two different ways to format form parameters! Instead, they actually mean two entirely different things. The GET method means, “I believe that there is a document at this URL; please return it.” The POST method means, “Here is an action that I want performed.”

Note that POST must always be the method used for actions on the Web that have side effects. Fetching a URL with GET should never produce any change in the web site from which the page is fetched. Requests submitted with POST, by contrast, can be requests to add, delete, or alter content.

Successful Form POSTs Should Always Redirect

You will already have noticed that the POST we performed earlier in this chapter did something very interesting: instead of simply returning a status of 200 followed by a page of weather forecast data, it instead returned a 302 redirect that urllib2 obeyed by performing a GET for the page named in the Location: header. Why add this extra level of indirection, instead of just returning a useful page?

The answer is that a web site leaves users in a very difficult position if it answers a POST form submission with a literal web page. You will probably recognize these symptoms:

- The web browser will display the URL to which the POST was made, which is generally fairly generic; however, the actual page content will be something quite specific. For example, had the query in the previous section not performed its redirect, then a user of the form would wind up at the URL /zipcity.php. This sounds very general, but the user would be looking at the specific forecast for Atlanta.
- The URL winds up being useless when bookmarked or shared. Because it was the form parameters that brought Atlanta up, someone e-mailing the /zipcity.php URL to a friend would send them to a page that displays an error instead. For example, when the /zipcity.php URL is visited without going through the form, the NWS web site displays this message: "Nothing was entered in the search box, or an incorrect format was used."
- The user cannot reload the web page without receiving a frightening warning about whether he wants to repeat his action. This is because, to refetch the page, his browser would have to resubmit the POST. Per the semantics we discussed previously, a POST represents some action that might be dangerous; destructive; or, at the very least, repetitive (the user might wind up generating several copies of the same tweet or something) if issued several times. Often, a POST that deletes an item can only succeed once, and it will show an error page when reloaded.

For all of these reasons, well-designed user-facing POST forms always redirect to a page that shows the result of the action, and this page can be safely bookmarked, shared, stored, and reloaded. This is an important feature of modern browsers: if a POST results in a redirect, then pressing the reload button simply refetches the final URL and does *not* reattempt the whole train of redirects that lead to the current location!

The one exception is that an unsuccessful POST should immediately display the form again, with its fields already filled out—do not make the user type everything again!—and with their errors or omissions marked, so that the user can correct them. The reason that a redirect is not appropriate here is that, unless the POST parameters are saved somewhere by the web server, the server will not know how to fill out the form (or what errors to flag) when the GET arrives a few moments later from the redirected browser.

Note that early browsers interpreted a 302 response inconsistently, so code 303 was created to unambiguously request the right behavior in response to a POST. There seems to be fear among some web developers that some ancient browsers might not understand 303; however, I have never actually seen any specific browsers named that are still in use that will not interpret this more-correct HTTP response code correctly.

POST And APIs

Almost none of the caveats given in the last two sections apply when an HTTP `POST` is designed for consumption by a program other than a web browser. This is because all of the issues that hinge upon user interaction, browser history, and the "reload" and "back" buttons will simply not apply.

To begin with, a `POST` designed for use by a program need not use the awkward `x-www-form-urlencoded` data format for its input parameters. Instead, it can specify any combination of content type and input data that its programmer is prepared for it to handle. Data formats like XML, JSON, and BSON are often used. Some services even allow entire documents or images to be posted raw, so long as the request `Content-Type:` header is set to correctly indicate their type.

The next common difference is that API calls made through POST rarely result in redirection; sending a program to another URL to receive the result of such calls (and thus requiring the client to make a second round-trip to the server to perform that download) only makes sense if the whole point of the service is to map requests into references to other URLs.

Finally, the range of payload content types returned from API calls is much broader than the kinds of data that can usefully be returned to browsers. Instead of supporting only things like web pages, style sheets, and images, the programs that consume web APIs often welcome rich formatted data like that supported by formats like XML and JSON. Often, a service will choose the same data format for both its `POST` request and return values, and thus require client programs to use only one data library for coercion, rather than two.

Note that many API services are designed for use with a JavaScript program running inside of a web page delivered through a normal `GET` call. Despite the fact that the JavaScript is running in a browser, such services will act like APIs rather than user form posts: they typically do not redirect, but instead send and receive data payloads (typically) rather than browsable web pages.

REST And More HTTP Methods

We have just introduced the topic of web-based APIs, which fetch documents and data using `GET` and `POST` to specific URLs. Therefore, we should immediately note that many modern web services try to integrate their APIs more tightly with HTTP by going beyond the two most common HTTP methods by implementing additional methods like `PUT` and `DELETE`.

In general, a web API that is implemented entirely with `POST` commands remains opaque to proxies, caches, and any other tools that support the HTTP protocol. All they know is that a series of unrepeatable special commands are passing between the client and the server. But they cannot detect whether resources are being queried, created, destroyed, or manipulated.

A design pattern named "Representational State Transfer" has therefore been taking hold in many developer communities. This design pattern is based on Roy Fielding's celebrated 2000 doctoral dissertation that first fully defined the concept. It specifies that the *nouns* of an API should live at their own URLs. For example, `PUT`, `GET`, `POST`, and `DELETE` should be used, respectively, to create, fetch, modify, and remove the documents living at these URLs.

By coupling this basic recommendation with further guidelines, the REST methodology guides the creation of web services that make more complete use of the HTTP protocol (instead of treating it as a dumb transport mechanism). Such web services also offer quite clean semantics, and can be accelerated by the same caching proxies that are often used to speed the delivery of normal web pages.

There are now entire books dedicated to RESTful web services, which I recommend you peruse if you are going to be building programmer web interfaces in the future!

Note that HTTP supports arbitrary method names, even though the standard defines specific semantics for `GET` and `POST` and all of the rest. Tradition would dictate using the well-known methods defined in the standard unless you are using a specific framework or methodology that recognizes and has defined other methods.

Identifying User Agents and Web Servers

You may have noticed that the HTTP request we opened the chapter with advertised the fact that it was generated by a Python program:

```
User-Agent: Python-urllib/2.6
```

This header is optional in the HTTP protocol, and many sites simply ignore or log it. It can be useful when sites want to know which browsers their visitors use most often, and it can sometimes be used to distinguish search engine spiders (*bots*) from normal users browsing a site. For example, here are a few of the user agents that have hit my own web site in the past few minutes:

```
Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)
Mozilla/5.0 (compatible; YandexBot/3.0; +http://yandex.com/bots)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
»   1.1.4322; .NET CLR 2.0.50727)
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.3
»   (KHTML, like Gecko) Chrome/6.0.472.62 Safari/534.3
```

You will note that, the `urllib2` user agent string notwithstanding, most clients choose to identify themselves as some form of the original Netscape browser, whose internal code name was `Mozilla`. But then, in parentheses, these same browsers secretly admit that they are really some other kind of browser.

Many web sites are sensitive to the kinds of browsers that view them, most often because their designers were too lazy to make the sites work with anything other than Internet Explorer. If you need to access such sites with `urllib2`, you can simply instruct it to lie about its identity, and the receiving web site will not know the difference:

```
>>> url = 'https://wca.eclaim.com/'
>>> urllib2.urlopen(url).read()
'<HTML>...The following are...required...Microsoft Internet Explorer...'
>>> agent = 'Mozilla/5.0 (Windows; U; MSIE 7.0; Windows NT 6.0; en-US)'
>>> request = urllib2.Request(url)
>>> request.add_header('User-Agent', agent)
>>> urllib2.urlopen(request).read()
'\r\n<HTML>\r\n<HEAD>\r\n\t<TITLE>Eclaim.com - Log In</TITLE>...'
```

There are databases of possible user agent strings online at several sites that you can reference both when analyzing agent strings that your own servers have received, as well as when concocting strings for your own HTTP requests:

```
http://www.zytrax.com/tech/web/browser_ids.htm
http://www.useragentstring.com/pages/useragentstring.php
```

Besides using the agent string to enforce compatibility requirements—usually in an effort to reduce development and support costs—some web sites have started using the string to detect mobile browsers and redirect the user to a miniaturized mobile version of the site for better viewing on phones and iPods. A Python project named `mobile.sniffer` that attempts to support this technique can be found on the Package Index.

Content Type Negotiation

It is always possible to simply make an HTTP request and let the server return a document with whatever `Content-Type:` is appropriate for the information we have requested. Some of the usual content types encountered by a browser include the following:

```
text/html
text/plain
text/css
image/gif
image/jpeg
image/x-png
application/javascript
application/pdf
application/zip
```

If the web service is returning a generic data stream of bytes that it cannot describe more specifically, it can always fall back to the content type:

```
application/octet-stream
```

But some clients do support all content types. Such clients like to encourage servers to send compatible content when several versions of a resource are available. This selection can occur along several axes: older browsers might not know about new, up-and-coming image formats; some browsers can only read certain encodings; and, of course, each user has particular languages that she can read and prefers web sites to deliver content in her native tongue, if possible.

Consult RFC 2616 if you find that your Python web client is sophisticated enough that you need to wade into content negotiation. The four headers that will interest you include the following:

```
Accept
Accept-Charset
Accept-Language
Accept-Encoding
```

Each of these headers supports a comma-separated list of items, where each item can be given a weight between one and zero (larger weights indicate more preferred items) by adding a suffix that consists of a semi-colon and `q=` string to the item. The result will look something like this (using, for illustration, the `Accept:` header that my Google Chrome browser seems to be currently using):

```
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;
»   q=0.8,image/png,*/*;q=0.5
```

This indicates that Chrome prefers XML and XHTML, but will accept HTML or even plain text if those are the only document formats available; that Chrome prefers PNG images when it can get them; and that it has no preference between all of the other content types in existence.

The HTTP standard also describes the possibility of a client receiving a `300` "Multiple Choices" response and getting to choose its own content type; however, this does not seem to be a widely-implemented mechanism, and I refer you to the RFC should you ever need to use it.

Compression

While many documents delivered over HTTP are already fairly heavily compressed, including images (so long as they are not raw TIFF or BMP) and file formats like PDF (at the option of the document author), web pages themselves are written in verbose SGML dialects (see Chapter 10) that can consume much less bandwidth if subjected to generic textual compression. Similarly, CSS and JavaScript files also contain very stereotyped patterns of punctuation and repeated variable names, which is very amenable to compression.

Web clients can make servers aware that they accept compressed documents by listing the formats they support in a request header, as in this example:

```
Accept-Encoding: gzip
```

For some reason, many sites seem to not offer compression unless the `User-Agent:` header specifies something they recognize. Thus, to convince Google to compress its Google News page, you have to use `urllib2` something like this:

```
>>> request = urllib2.Request('http://news.google.com/')
>>> request.add_header('Accept-Encoding', 'gzip')
>>> request.add_header('User-Agent', 'Mozilla/5.0')
>>> info = opener.open(request)
--------------------------------------------------
GET / HTTP/1.1
Host: news.google.com
User-Agent: Mozilla/5.0
Connection: close
Accept-Encoding: gzip
-------------------- Response --------------------
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
...
Content-Encoding: gzip
...
```

Remember that web servers do not have to perform compression, and that many will ignore your `Accept-Encoding:` header. Therefore, you should always check the content encoding of the response, and perform decompression only when the server declares that it is necessary:

```
>>> info.headers['Content-Encoding'] == 'gzip'
True
>>> import gzip, StringIO
>>> gzip.GzipFile(fileobj=StringIO.StringIO(info.read())).read()
'<!DOCTYPE HTML ...<html>...</html>'
```

As you can see, Python does not let us pass the file-like `info` response object directly to the `GzipFile` class because, alas, it lacks a `tell()` method. In other words, it is not quite file-like enough. Here, we can perform the quick work-around of reading the whole compressed file into memory and then wrapping it in a `StringIO` object that does support `tell()`.

HTTP Caching

Many elements of a typical web site design are repeated on every page you visit, and your browsing would slow to a crawl if every image and decoration had to be downloaded separately for every page you viewed. Well-configured web servers therefore add headers to every HTTP response that allow browsers, as well as any proxy caches between the browser and the server, to continue using a copy of a downloaded resource for some period of time until it expires.

You might think that adding a simple expiration date to each resource that could be cached and redisplayed would have been a sufficient innovation. However, given the real-world behaviors of servers, caches, and browsers, it was prudent for the HTTP specification to detail a much more complicated scheme involving several interacting headers. Several pages are expended, for example, on the specific question of how to determine how old a cached copy of a page is. I refer you to RFC 2616 for the real details, but I will cover a few of the most common cases here.

There are two basic mechanisms by which servers can support client caching.

In the first approach, an HTTP response includes an `Expires:` header that formats a date and time using the same format as the standard `Date:` header:

```
Expires: Sun, 21 Jan 2010 17:06:12 GMT
```

However, this requires the client to check its clock—and many computers run clocks that are far ahead of or behind the real current date and time.

This brings us to a second, more modern alternative, the `Cache-Control` header, that depends only on the client being able to correctly count seconds forward from the present. For example, to allow an image or page to be cached for an hour but then insist that it be refetched once the hour is up, a cache control header could be supplied like this:

```
Cache-Control: max-age=3600, must-revalidate
```

When the time comes to *validate* a cached resource, HTTP offers a very nice shortcut: the client can ask the server to retransmit the resource only if a new version has indeed been released. There are two fields that the client can supply. Either content type is sufficient to convince most servers to answer with only an HTTP header, but no content type or body, if the cached resource is still current. One possibility is to send back the value that the `Last-modified:` header had in the HTTP response that first requested the item:

```
If-Modified-Since: Sun, 21 Jan 2010 14:06:12 GMT
```

Alternatively, if the server tagged the resource version with a hash or version identifier in an `Etag:` header—either approach will work, so long as the value always changes between versions of the resource—then the client can send that value back:

```
Etag: BFDS2Cpq/BM6w
```

Note that all of this depends on getting some level of cooperation from the server. If a web server fails to provide any caching guidelines and also does not supply either a `Last-modified:` or `Etag:` header for a particular resource, then clients have no choice but to fetch the resource every time it needs to be displayed to a user.

Caching is such a powerful technology that many web sites go ahead and put HTTP caches like Squid or Varnish in front of their server farms, so that frequent requests for the most popular parts of their site can be answered without loading down the main servers. Deploying caches geographically can also save bandwidth. In a celebrated question-and-answer session with the readers of Reddit about The Onion's then-recent migration to Django, the site maintainers—who use a content delivery network (CDN) to transparently serve local caches of The Onion's web site all over the world—indicated that they were able to reduce their server load by two-thirds by asking the CDN to cache 404 errors! You can read

the report here: http://www.reddit.com/r/django/comments/bhvhz/the_onion_uses_django_and_why_it_matters_to_us/

Note that web caches also have to worry about invalidating web resources that are hit by a POST, PUT, or DELETE request because any of those operations could presumably change the data that will be returned to users from that resource. Caching proxies are tricky things to write and require a vast attention span with respect to reading standards!

Neither urllib2 nor mechanize seem to support caching; so if you need a local cache, you might want to look at the httplib2 module available on the Python Package Index.

The HEAD Method

It's possible that you might want your program to check a series of links for validity or whether they have moved, but you do not want to incur the expense of actually downloading the body that would follow the HTTP headers. In this case, you can issue a HEAD request. This is directly possible through httplib, but it can also be performed by urllib2 if you are willing to write a small request class of your own:

```
>>> class HeadRequest(urllib2.Request):
...     def get_method(self):
...         return 'HEAD'
>>> info = urllib2.urlopen(HeadRequest('http://www.google.com/'))
>>> info.read()
''
```

You can see that the body of the response is completely empty.

HTTPS Encryption

With the processors of the late 1990s, the prospect of turning on encryption for a web site was a very expensive one; I remember that at least one vendor even made accelerator cards that would do SSL computations in hardware. But the great gulf that Moore's law has opened between processor speed and the other subsystems on a computer means that there is no reason not to deploy SSL everywhere that user data or identity needs protection. When Google moved its GMail service to being HTTPS-only, the company asserted that the certificate and encryption routines were only adding a few percent to the server CPU usage.

An encrypted URL starts with https: instead of simply http:, uses the default port 443 instead of port 80, and uses TLS; review Chapter 6 to remember how TLS/SSL operates.

Encryption places web servers in a dilemma: encryption has to be negotiated before the user can send his HTTP request, lest all of the information in it be divulged; but until the request is transmitted, the server does not know what Host: the request will specify. Therefore, encrypted web sites still live under the old problem of having to use a different IP address for every domain that must be hosted.

A technique known as "Server Name Indication" (SNI) has been developed to get around this traditional restriction; however, Python does not yet support it. It appears, though, that a patch was applied to the Python 3 trunk with this feature, only days prior to the time of writing. Here is the ticket in case you want to follow the issue: http://bugs.python.org/issue5639

Hopefully, there will be a Python 3 edition of this book within the next year or two that will be able to happily report that SNI is fully supported by urllib2!

To use HTTPS from Python, simply supply an https: method in your URL:

```
>>> info = urllib2.urlopen('https://www.ietf.org/rfc/rfc2616.txt')
```

If the connection works properly, then neither your government nor any of the various large and shadowy corporations that track such things should be able to easily determine either the search term you used or the results you viewed.

HTTP Authentication

The HTTP protocol came with a means of authentication that was so poorly thought out and so badly implemented that it seems to have been almost entirely abandoned. When a server was asked for a page to which access was restricted, it was supposed to return a response code:

```
HTTP/1.1 401 Authorization Required
...
WWW-Authenticate: Basic realm="voetbal"
...
```

This indicated that the server did not know who was requesting the resource, so it could not decide whether to grant permission. By asking for `Basic` authentication, the site would induce the web browser to pop up a dialog box asking for a username and password. The information entered would then be sent back in a header as part of a second request for exactly the same resource. The authentication token was generated by doing base64 encoding on the colon-separated username and password:

```
>>> import base64
>>> print base64.b64encode("guido:vanOranje!")
Z3VpZG86dmFuT3JhbmplIQ==
```

This, of course, just protects any special characters in the username and password that might have been confused as part of the headers themselves; it does *not* protect the username and password at all, since they can very simply be decoded again:

```
>>> print base64.b64decode("Z3VpZG86dmFuT3JhbmplIQ==")
guido:vanOranje!
```

Anyway, once the encoded value was computed, it could be included in the second request like this:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

An incorrect password or unknown user would elicit additional `401` errors from the server, resulting in the pop-up box appearing again and again. Finally, if the user got it right, she would either be shown the resource or—if she in fact did not have permission—be shown a response code like the following:

```
403 Forbidden
```

Python supports this kind of authentication through a handler that, as your program uses it, can accumulate a list of passwords. It is very careful to keep straight which passwords go with which web sites, lest it send the wrong one and allow one web site operator to learn your password to another site! It also checks the `realm` string specified by the server in its `WWW-Authenticate` header; this allows a single web site to have several separate areas inside that each take their own set of usernames and passwords.

The handler can be created and populated with a single password like this:

```
auth_handler =  .HTTPBasicAuthHandler()
auth_handler.add_password(realm='voetbal', uri='http://www.onsoranje.nl/',
                          user='guido', passwd='vanOranje!')
```

The resulting handler can be passed into `build_opener()`, just as we did with our debugging handler early in this chapter.

Concern over revealing passwords lead to the development of "digest authentication" in the late 1990s; however, if you are going to support user authentication on a site, then you should probably go all the way and use HTTPS so that everything gets protected, plain-text passwords and all. See the documentation for the `HTTPDigestAuthHandler` in `urllib2` if you need to write a client that supports it.

Unfortunately, browser support for any kind of HTTP authentication is very poor—most do not even provide a logout button!—so you should avoid designing sites that use these mechanisms. We will learn about the modern alternative in the next section.

Cookies

The actual mechanism that powers user identity tracking, logging in, and logging out of modern web sites is the cookie. The HTTP responses sent by a server can optionally include a number of `Set-cookie:` headers that browsers store on behalf of the user. In every subsequent request made to that site—or even to any of its sub-domains, if the cookie allows that—the browser will include a `Cookie:` header corresponding to each cookie that has been set.

How can cookies be used?

The most obvious use is to keep up with user identity. To support logging in, a web site can deploy a normal form that asks for your username and password (or e-mail address and password, or whatever). If the form is submitted successfully, then the response can include a cookie that says, "this request is from the user Ken." Every subsequent request that the browser makes for a document, image, or anything else under that domain will include the cookie and let the site know who is requesting it. And finally, a "Log out" button can be provided that clears the cookie from the browser.

Obviously, the cookie cannot really be formatted so it just baldly asserts a user's identity because users would figure this out and start writing their own cookies that let them assume other user identities. Therefore, one of following two approaches is used in practice:

- The server can store a random unguessable value in the cookie that also gets written to its back-end database. Incoming cookies are then checked against the database. Sessions can be made to time out by deleting entries from this database once they reach a certain age.
- The cookie can be a block of data that is encrypted with a secret key held only by the web service. Upon decryption, it would contain a user identifier and a timestamp that prevented it from being honored if it were too old.

Cookies can also be used for feats other than simply identifying users. For example, a site can issue a cookie to every browser that connects, enabling it to track even casual visitors. This approach enables an online store to let visitors start building a shopping cart full of items—and even check out and complete their purchase—without ever being forced to create an account. Since most e-commerce sites also like to support accounts for the convenience of returning customers, they may also need to program their servers to support merging a temporary shopping cart with a permanent per-customer shopping cart in case someone arrives, selects several items, and then logs in and winds up being an already-existing user.

From the point of view of a web client, cookies are moderately short strings that have to be stored and then divulged when matching requests are made. The Python Standard Library puts this logic in its own module, `cookielib`, whose `CookieJar` objects can be used as small cookie databases by the `HTTPCookieProcessor` in `urllib2`. To see its effect, you need go no further than the front page of Google, which sets cookies in the mere event of an unknown visitor arriving at the site for the first time. Here is how we create a new opener that knows about cookies:

```
>>> import cookielib
>>> cj = cookielib.CookieJar()
>>> cookie_opener = urllib2.build_opener(VerboseHTTPHandler,
...   urllib2.HTTPCookieProcessor(cj))
```

Opening the Google front page will result in two different cookies getting set:

```
>>> response = cookie_opener.open('http://www.google.com/')
--------------------------------------------------
GET / HTTP/1.1
...
-------------------- Response --------------------
HTTP/1.1 200 OK
...
Set-Cookie: PREF=ID=94381994af6d5c77:FF=0:TM=1288205983:LM=1288205983:S=Mtwivl7EB73uL5Ky;↵
 expires=Fri, 26-Oct-2012 18:59:43 GMT; path=/; domain=.google.com
Set-Cookie: NID=40=rWLn_I8_PAhUF62J0yFLtb1-AoftgU0RvGSsa81FhTvd4vXD91iU5DOEdxSVt4otiISY-↵
3RfEYcGFHZA52w3-85p-hujagtB9akaLnS0QHEt2v8lkkelEGbpo7oWr9u5; expires=Thu, 28-Apr-2011↵
 18:59:43 GMT; path=/; domain=.google.com; HttpOnly
...
```

If you consult the `cookielib` documentation, you will find that you can do more than query and modify the cookies that have been set. You can also automatically store them in a file, so that they survive from one Python session to the next. You can even create cookie processors that implement your own custom policies with respect to which cookies to store and which to divulge.

Note that if we visit another Google page—the options page, for example—then both of the cookies set previously get submitted in the same `Cookie` header, separated by a semicolon:

```
>>> response = cookie_opener.open('http://www.google.com/intl/en/options/')
--------------------------------------------------
GET /intl/en/options/ HTTP/1.1
...
Cookie: PREF=ID=94381994af6d5c77:FF=0:TM=1288205983:LM=1288205983:S=Mtwivl7EB73uL5Ky;↵
 NID=40=rWLn_I8_PAhUF62J0yFLtb1-AoftgU0RvGSsa81FhTvd4vXD91iU5DOEdxSVt4otiISY-↵
3RfEYcGFHZA52w3-85p-hujagtB9akaLnS0QHEt2v8lkkelEGbpo7oWr9u5
...
-------------------- Response --------------------
HTTP/1.1 200 OK
...
```

Servers can constrain a cookie to a particular `domain` and `path`, in addition to setting a `Max-age` or expires time. Unfortunately, some browsers ignore this setting, so sites should never base their security on the assumption that the expires time will be obeyed. Therefore, servers can mark cookies as `secure`; this ensures that such cookies are only transmitted with HTTPS requests to the site and never in unsecure HTTP requests. We will see uses for this in the next session.

Some browsers also obey a non-standard `HttpOnly` flag, which you can see in one of the Google cookies shown a moment ago. This flag hides the cookie from any JavaScript programs running on a web page. This is an attempt to make cross-site scripting attacks more difficult, as we will soon learn.

Note that there are other mechanisms besides cookies available if a particularly aggressive domain wants to keep track of your activities; many of the best ideas have been combined in a project called "evercookie": `http://samy.pl/evercookie/`

I do not recommend using these approaches in your own applications; instead, I recommend using standard cookies, so that intelligent users have at least a chance at opting to control your monitoring!

But you should know that these other mechanisms exist if you are writing web clients, proxies, or even if you simply browse the Web yourself and are interested in controlling your identity.

HTTP Session Hijacking

A perpetual problem with cookies is that web site designers do not seem to realize that cookies need to be protected as zealously as your username and password. While it is true that well-designed cookies expire and will no longer be accepted as valid by the server, cookies—while they last—give exactly as much access to a web site as a username and password. If someone can make requests to a site with your login cookie, the site will think it is you who has just logged in.

Some sites do not protect cookies at all: they might require HTTPS for your username and password, but then return you to normal HTTP for the rest of your session. And with every HTTP request, your session cookies are transmitted in the clear for anyone to intercept and start using.

Other sites are smart enough to protect subsequent page loads with HTTPS, even after you have left the login page, but they forget that static data from the same domain, like images, decorations, CSS files, and JavaScript source code, will also carry your cookie. The better alternatives are to either send all of that information over HTTPS, or to carefully serve it from a different domain or path that is outside the jurisdiction of the session cookie.

And despite the fact this problem has existed for years, at the time of writing it is once again back in the news with the celebrated release of Firesheep. Sites need to learn that session cookies should always be marked as `secure`, so that browsers will not divulge them over insecure links.

Earlier generations of browsers would refuse to cache content that came in over HTTPS, and that might be where some developers got into the habit of not encrypting most of their web site. But modern browsers will happily cache resources fetched over HTTPS—some will even save it on disk if the `Cache-control:` header is set to `public`—so there are no longer good reasons not to encrypt everything sent from a web site. Remember: If your users really need privacy, then exposing even what images, decorations, and JavaScript they are downloading might allow an observer to guess which pages they are visiting and which actions they are taking on your site.

Should you happen to observe or capture a `Cookie:` header from an HTTP request that you observe, remember that there is no need to store it in a `CookieJar` or represent it as a `cookielib` object at all. Indeed, you could not do that anyway because the outgoing `Cookie:` header does not reveal the `domain` and `path` rules that the cookie was stored with. Instead, just inject the `Cookie:` header raw into the requests you make to the web site:

```
request = urllib2.Request(url)
request.add_header('Cookie', intercepted_value)
info = urllib2.urlopen(request)
```

As always, use your powers for good and not evil!

Cross-Site Scripting Attacks

The earliest experiments with scripts that could run in web browsers revealed a problem: all of the HTTP requests made by the browser were done with the authority of the user's cookies, so pages could cause quite a bit of trouble by attempting to, say, `POST` to the online web site of a popular bank asking that money be transferred to the attacker's account. Anyone who visited the problem site while logged on to that particular bank in another window could lose money.

To address this, browsers imposed the restriction that scripts in languages like JavaScript can only make connections back to the site that served the web page, and not to other web sites. This is called the "same origin policy."

So the techniques to attack sites have evolved and mutated. Today, would-be attackers find ways around this policy by using a constellation of attacks called *cross-site scripting* (known by the acronym *XSS* to prevent confusion with *Cascading Style Sheets*). These techniques include things like finding the fields on a web page where the site will include snippets of user-provided data without properly escaping them, and then figuring out how to craft a snippet of data that will perform some compromising action on behalf of the user or send private information to a third party. Next, the would-be attackers release a link or code containing that snippet onto a popular web site, bulletin board, or in spam e-mails, hoping that thousands of people will click and inadvertently assist in their attack against the site.

There are a collection of techniques that are important for avoiding cross-site scripting; you can find them in any good reference on web development. The most important ones include the following:

- When processing a form that is supposed to submit a `POST` request, always carefully disregard any `GET` parameters.
- Never support URLs that produce some side effect or perform some action simply through being the subject of a `GET`.
- In every form, include not only the obvious information—such as a dollar amount and destination account number for bank transfers—but also a hidden field with a secret value that must match for the submission to be valid. That way, random `POST` requests that attackers generate with the dollar amount and destination account number will not work because they will lack the secret that would make the submission valid.

While the possibilities for XSS are not, strictly speaking, problems or issues with the HTTP protocol itself, it helps to have a solid understanding of them when you are trying to write any program that operates safely on the World Wide Web.

WebOb

We have seen that HTTP requests and responses are each represented by ad-hoc objects in `urllib2`. Many Python programmers find its interface unwieldy, as well as incomplete! But, in their defense, the objects seem to have been created as minimal constructs, containing only what `urllib2` needed to function.

But a library called WebOb is also available for Python (and listed on the Python Package Index) that contains HTTP request and response classes that were designed from the other direction: that is, they were intended all along as general-purpose representations of HTTP in all of its low-level details. You can learn more about them at the WebOb project web page: `http://pythonpaste.org/webob/`

This library's objects are specifically designed to interface well with WSGI, which makes them useful when writing HTTP servers, as we will see in Chapter 11.

Summary

The HTTP protocol sounds simple enough: each request names a document (which can be an image or program or whatever), and responses are supposed to supply its content. But the reality, of course, is rather more complicated, as its main features to support the modern Web have driven its specification, RFC 2616, to nearly 60,000 words. In this chapter, we tried to capture its essence in around 10,000 words and obviously had to leave things out. Along the way, we discussed (and showed sample Python code) for the following concepts:

- URLs and their structure.
- The GET method and fetching documents.
- How the Host: header makes up for the fact that the hostname from the URL is not included in the path that follows the word GET.
- The success and error codes returned in HTTP responses and how they induce browser actions like redirection.
- How persistent connections can increase the speed at which HTTP resources can be fetched.
- The POST method for performing actions and submitting forms.
- How redirection should always follow the successful POST of a web form.
- That POST is often used for web service requests from programs and can directly return useful information.
- Other HTTP methods exist and can be used to design web-centric applications using a methodology called REST.
- Browsers identify themselves through a user agent string, and some servers are sensitive to this value.
- Requests often specify what content types a client can display, and well-written servers will try to choose content representations that fit these constraints.
- Clients can request—and servers can use—compression that results in a page arriving more quickly over the network.
- Several headers and a set of rules govern which HTTP-delivered documents can and cannot be cached.
- The HEAD command only returns the headers.
- The HTTPS protocol adds TLS/SSL protection to HTTP.
- An old and awkward form of authentication is supported by HTTP itself.
- Most sites today supply their own login form and then use cookies to identify users as they move across the site.
- If a cookie is captured, it can allow an attacker to view a web site as though the attacker were the user whose cookie was stolen.
- Even more difficult classes of attack exist on the modern dynamic web, collectively called cross-site-scripting attacks.

Armed with the knowledge and examples in this chapter, you should be able to use the urllib2 module from the Standard Library to fetch resources from the Web and even implement primitive browser behaviors like retaining cookies.

CHAPTER 10

Screen Scraping

Most web sites are designed first and foremost for human eyes. While well-designed sites offer formal APIs by which you can construct Google maps, upload Flickr photos, or browse YouTube videos, many sites offer nothing but HTML pages formatted for humans. If you need a program to be able to fetch its data, then you will need the ability to dive into densely formatted markup and retrieve the information you need—a process known affectionately as screen scraping.

In one's haste to grab information from a web page sitting open in your browser in front of you, it can be easy for even experienced programmers to forget to check whether an API is provided for data that they need. So try to take a few minutes investigating the site in which you are interested to see if some more formal programming interface is offered to their services. Even an RSS feed can sometimes be easier to parse than a list of items on a full web page.

Also be careful to check for a "terms of service" document on each site. YouTube, for example, offers an API and, in return, disallows programs from trying to parse their web pages. Sites usually do this for very important reasons related to performance and usage patterns, so I recommend always obeying the terms of service and simply going elsewhere for your data if they prove too restrictive.

Regardless of whether terms of service exist, always try to be polite when hitting public web sites. Cache pages or data that you will need for several minutes or hours, rather than hitting their site needlessly over and over again. When developing your screen-scraping algorithm, test against a copy of their web page that you save to disk, instead of doing an HTTP round-trip with every test. And always be aware that excessive use can result in your IP being temporarily or permanently blocked from a site if its owners are sensitive to automated sources of load.

Fetching Web Pages

Before you can parse an HTML-formatted web page, you of course have to acquire some. Chapter 9 provides the kind of thorough introduction to the HTTP protocol that can help you figure out how to fetch information even from sites that require passwords or cookies. But, in brief, here are some options for downloading content.

- You can use `urllib2`, or the even lower-level `httplib`, to construct an HTTP request that will return a web page. For each form that has to be filled out, you will have to build a dictionary representing the field names and data values inside; unlike a real web browser, these libraries will give you no help in submitting forms.
- You can to install `mechanize` and write a program that fills out and submits web forms much as you would do when sitting in front of a web browser. The downside is that, to benefit from this automation, you will need to download the page containing the form HTML before you can then submit it—possibly doubling the number of web requests you perform!

- If you need to download and parse entire web sites, take a look at the Scrapy project, hosted at `http://scrapy.org`, which provides a framework for implementing your own web spiders. With the tools it provides, you can write programs that follow links to every page on a web site, tabulating the data you want extracted from each page.
- When web pages wind up being incomplete because they use dynamic JavaScript to load data that you need, you can use the `QtWebKit` module of the `PyQt4` library to load a page, let the JavaScript run, and then save or parse the resulting complete HTML page.
- Finally, if you really need a browser to load the site, both the Selenium and Windmill test platforms provide a way to drive a standard web browser from inside a Python program. You can start the browser up, direct it to the page of interest, fill out and submit forms, do whatever else is necessary to bring up the data you need, and then pull the resulting information directly from the DOM elements that hold them.

These last two options both require third-party components or Python modules that are built against large libraries, and so we will not cover them here, in favor of techniques that require only pure Python.

For our examples in this chapter, we will use the site of the United States National Weather Service, which lives here: `www.weather.gov/`.

Among the better features of the United States government is its having long ago decreed that all publications produced by their agencies are public domain. This means, happily, that I can pull all sorts of data from their web site and not worry about the fact that copies of the data are working their way into this book.

Of course, web sites change, so the source code package for this book available from the Apress web site will include the downloaded pages on which the scripts in this chapter are designed to work. That way, even if their site undergoes a major redesign, you will still be able to try out the code examples in the future. And, anyway—as I recommended previously—you should be kind to web sites by always developing your scraping code against a downloaded copy of a web page to help reduce their load.

Downloading Pages Through Form Submission

The task of grabbing information from a web site usually starts by reading it carefully with a web browser and finding a route to the information you need. Figure 10–1 shows the site of the National Weather Service; for our first example, we will write a program that takes a city and state as arguments and prints out the current conditions, temperature, and humidity. If you will explore the site a bit, you will find that city-specific forecasts can be visited by typing the city name into the small "Local forecast" form in the left margin.

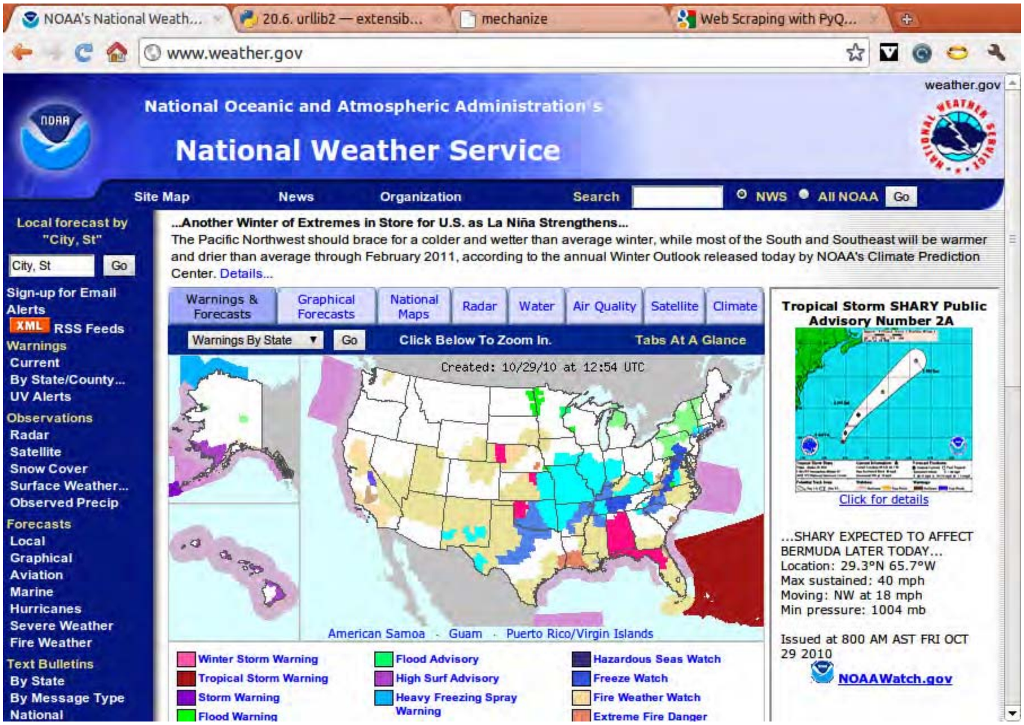


Figure 10–1. The National Weather Service web site

When using the urllib2 module from the Standard Library, you will have to read the web page HTML manually to find the form. You can use the View Source command in your browser, search for the words “Local forecast,” and find the following form in the middle of the sea of HTML:

```
<form method="post" action="http://forecast.weather.gov/zipcity.php" ...>
  ...
  <input type="text" id="zipcity" name="inputstring" size="9"
»   value="City, St" onfocus="this.value='';" />
  <input type="submit" name="Go2" value="Go" />
</form>
```

The only important elements here are the <form> itself and the <input> fields inside; everything else is just decoration intended to help human readers.

This form does a POST to a particular URL with, it appears, just one parameter: an inputstring giving the city name and state. Listing 10–1 shows a simple Python program that uses only the Standard Library to perform this interaction, and saves the result to phoenix.html.

Listing 10–1. Submitting a Form with “urllib2”

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 10 - fetch_urllib2.py
# Submitting a form and retrieving a page with urllib2

import urllib, urllib2
```

```
data = urllib.urlencode({'inputstring': 'Phoenix, AZ'})
info = urllib2.urlopen('http://forecast.weather.gov/zipcity.php', data)
content = info.read()
open('phoenix.html', 'w').write(content)
```

On the one hand, `urllib2` makes this interaction very convenient; we are able to download a forecast page using only a few lines of code. But, on the other hand, we had to read and understand the form ourselves instead of relying on an actual HTML parser to read it. The approach encouraged by `mechanize` is quite different: you need only the address of the opening page to get started, and the library itself will take responsibility for exploring the HTML and letting you know what forms are present. Here are the forms that it finds on this particular page:

```
>>> import mechanize
>>> br = mechanize.Browser()
>>> response = br.open('http://www.weather.gov/')
>>> for form in br.forms():
...     print '%r %r %s' % (form.name, form.attrs.get('id'), form.action)
...     for control in form.controls:
...         print '   ', control.type, control.name, repr(control.value)
None None http://search.usa.gov/search
»   hidden v:project 'firstgov'
»   text query ''
»   radio affiliate ['nws.noaa.gov']
»   submit None 'Go'
None None http://forecast.weather.gov/zipcity.php
»   text inputstring 'City, St'
»   submit Go2 'Go'
'jump' 'jump' http://www.weather.gov/
»   select menu ['http://www.weather.gov/alerts-beta/']
»   button None None
```

Here, `mechanize` has helped us avoid reading any HTML at all. Of course, pages with very obscure form names and fields might make it very difficult to look at a list of forms like this and decide which is the form we see on the page that we want to submit; in those cases, inspecting the HTML ourselves can be helpful, or—if you use Google Chrome, or Firefox with Firebug installed—right-clicking the form and selecting "Inspect Element" to jump right to its element in the document tree.

Once we have determined that we need the `zipcity.php` form, we can write a program like that shown in Listing 10–2. You can see that at no point does it build a set of form fields manually itself, as was necessary in our previous listing. Instead, it simply loads the front page, sets the one field value that we care about, and then presses the form's submit button. Note that since this HTML form did not specify a name, we had to create our own filter function—the lambda function in the listing—to choose which of the three forms we wanted.

Listing 10–2. *Submitting a Form with `mechanize`*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 10 - fetch_mechanize.py
# Submitting a form and retrieving a page with mechanize

import mechanize
br = mechanize.Browser()
br.open('http://www.weather.gov/')
br.select_form(predicate=lambda(form): 'zipcity' in form.action)
br['inputstring'] = 'Phoenix, AZ'
response = br.submit()
```

```
content = response.read()
open('phoenix.html', 'w').write(content)
```

Many `mechanize` users instead choose to select forms by the order in which they appear in the page—in which case we could have called `select_form(nr=1)`. But I prefer not to rely on the order, since the real identity of a form is inherent in the action that it performs, not its location on a page.

You will see immediately the problem with using `mechanize` for this kind of simple task: whereas Listing 10–1 was able to fetch the page we wanted with a single HTTP request, Listing 10–2 requires two round-trips to the web site to do the same task. For this reason, I avoid using `mechanize` for simple form submission. Instead, I keep it in reserve for the task at which it really shines: logging on to web sites like banks, which set cookies when you first arrive at their front page and require those cookies to be present as you log in and browse your accounts. Since these web sessions require a visit to the front page anyway, no extra round-trips are incurred by using `mechanize`.

The Structure of Web Pages

There is a veritable glut of online guides and published books on the subject of HTML, but a few notes about the format would seem to be appropriate here for users who might be encountering the format for the first time.

The Hypertext Markup Language (HTML) is one of many markup dialects built atop the Standard Generalized Markup Language (SGML), which bequeathed to the world the idea of using thousands of angle brackets to mark up plain text. Inserting bold and italics into a format like HTML is as simple as typing eight angle brackets:

```
The <b>very</b> strange book <i>Tristram Shandy</i>.
```

In the terminology of SGML, the strings `<b>` and `</b>` are each tags—they are, in fact, an opening and a closing tag—and together they create an element that contains the text `very` inside it. Elements can contain text as well as other elements, and can define a series of key/value attribute pairs that give more information about the element:

```
<p content="personal">I am reading <i document="play">Hamlet</i>.</p>
```

There is a whole subfamily of markup languages based on the simpler Extensible Markup Language (XML), which takes SGML and removes most of its special cases and features to produce documents that can be generated and parsed without knowing their structure ahead of time. The problem with SGML languages in this regard—and HTML is one particular example—is that they expect parsers to know the rules about which elements can be nested inside which other elements, and this leads to constructions like this unordered list `<ul>`, inside which are several list items `<li>`:

```
<ul><li>First<li>Second<li>Third<li>Fourth</ul>
```

At first this might look like a series of `<li>` elements that are more and more deeply nested, so that the final word here is four list elements deep. But since HTML in fact says that `<li>` elements cannot nest, an HTML parser will understand the foregoing snippet to be equivalent to this more explicit XML string:

```
<ul><li>First</li><li>Second</li><li>Third</li><li>Fourth</li></ul>
```

And beyond this implicit understanding of HTML that a parser must possess are the twin problems that, first, various browsers over the years have varied wildly in how well they can reconstruct the document structure when given very concise or even deeply broken HTML; and, second, most web page authors judge the quality of their HTML by whether their browser of choice renders it correctly. This has resulted not only in a World Wide Web that is full of sites with invalid and broken HTML markup, but

also in the fact that the permissiveness built into browsers has encouraged different flavors of broken HTML among their different user groups.

If HTML is a new concept to you, you can find abundant resources online. Here are a few documents that have been longstanding resources in helping programmers learn the format:

```
www.w3.org/MarkUp/Guide/
www.w3.org/MarkUp/Guide/Advanced.html
www.w3.org/MarkUp/Guide/Style
```

The brief bare-bones guide, and the long and verbose HTML standard itself, are good resources to have when trying to remember an element name or the name of a particular attribute value:

```
http://werbach.com/barebones/barebones.html
http://www.w3.org/TR/REC-html40/
```

When building your own web pages, try to install a real HTML validator in your editor, IDE, or build process, or test your web site once it is online by submitting it to

```
http://validator.w3.org/
```

You might also want to consider using the tidy tool, which can also be integrated into an editor or build process:

```
http://tidy.sourceforge.net/
```

We will now turn to that weather forecast for Phoenix, Arizona, that we downloaded earlier using our scripts (note that we will avoid creating extra traffic for the NWS by running our experiments against this local file), and we will learn how to extract actual data from HTML.

Three Axes

Parsing HTML with Python requires three choices:

- The parser you will use to digest the HTML, and try to make sense of its tangle of opening and closing tags
- The API by which your Python program will access the tree of concentric elements that the parser built from its analysis of the HTML page
- What kinds of selectors you will be able to write to jump directly to the part of the page that interests you, instead of having to step into the hierarchy one element at a time

The issue of selectors is a very important one, because a well-written selector can unambiguously identify an HTML element that interests you without your having to touch any of the elements above it in the document tree. This can insulate your program from larger design changes that might be made to a web site; as long as the element you are selecting retains the same ID, name, or whatever other property you select it with, your program will still find it even if after the redesign it is several levels deeper in the document.

I should pause for a second to explain terms like “deeper,” and I think the concept will be clearest if we reconsider the unordered list that was quoted in the previous section. An experienced web developer looking at that list rearranges it in her head, so that this is what it looks like:

```
<ul>
  <li>First</li>
  <li>Second</li>
```

```
  <li>Third</li>
  <li>Fourth</li>
</ul>
```

Here the <ul> element is said to be a "parent" element of the individual list items, which "wraps" them and which is one level "above" them in the whole document. The <li> elements are "siblings" of one another; each is a "child" of the <ul> element that "contains" them, and they sit "below" their parent in the larger document tree. This kind of spatial thinking winds up being very important for working your way into a document through an API.

In brief, here are your choices along each of the three axes that were just listed:

- The most powerful, flexible, and fastest parser at the moment appears to be the `HTMLParser` that comes with `lxml`; the next most powerful is the longtime favorite BeautifulSoup (I see that its author has, in his words, "abandoned" the new 3.1 version because it is weaker when given broken HTML, and recommends using the 3.0 series until he has time to release 3.2); and coming in dead last are the parsing classes included with the Python Standard Library, which no one seems to use for serious screen scraping.
- The best API for manipulating a tree of HTML elements is ElementTree, which has been brought into the Standard Library for use with the Standard Library parsers, and is also the API supported by `lxml`; BeautifulSoup supports an API peculiar to itself; and a pair of ancient, ugly, event-based interfaces to HTML still exist in the Python Standard Library.
- The `lxml` library supports two of the major industry-standard selectors: CSS selectors and XPath query language; BeautifulSoup has a selector system all its own, but one that is very powerful and has powered countless web-scraping programs over the years.

Given the foregoing range of options, I recommend using `lxml` when doing so is at all possible—installation requires compiling a C extension so that it can accelerate its parsing using `libxml2`—and using `BeautifulSoup` if you are on a machine where you can install only pure Python. Note that `lxml` is available as a pre-compiled package named `python-lxml` on Ubuntu machines, and that the best approach to installation is often this command line:

```
STATIC_DEPS=true pip install lxml
```

And if you consult the `lxml` documentation, you will find that it can optionally use the BeautifulSoup parser to build its own ElementTree-compliant trees of elements. This leaves very little reason to use BeautifulSoup by itself unless its selectors happen to be a perfect fit for your problem; we will discuss them later in this chapter.

But the state of the art may advance over the years, so be sure to consult its own documentation as well as recent blogs or Stack Overflow questions if you are having problems getting it to compile.

Diving into an HTML Document

The tree of objects that a parser creates from an HTML file is often called a Document Object Model, or DOM, even though this is officially the name of one particular API defined by the standards bodies and implemented by browsers for the use of JavaScript running on a web page.

The task we have set for ourselves, you will recall, is to find the current conditions, temperature, and humidity in the `phoenix.html` page that we have downloaded. You can view the page in full by downloading the source bundle for this book from Apress; I cannot include it verbatim here, because it

consists of nearly 17,000 characters of dense HTML code. But let me at least show you an excerpt: Listing 10–3, which focuses on the pane that we are interested in.

Listing 10–3. *Excerpt from the Phoenix Forecast Page*

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN"><html><head>
<title>7-Day Forecast for Latitude 33.45&deg;N and Longitude 112.07&deg;W (Elev. 1132
ft)</title><link rel="STYLESHEET" type="text/css" href="fonts/main.css">
...
<table cellspacing="0" cellspacing="0" border="0" width="100%"><tr align="center"><td><table
width='100%' border='0'>
<tr>
<td align ='center'>
<span class='blue1'>Phoenix, Phoenix Sky Harbor International Airport</span><br>
Last Update on 29 Oct 7:51 MST<br><br>
</td>
</tr>
<tr>
<td colspan='2'>
<table cellspacing='0' cellpadding='0' border='0' align='left'>
<tr>
<td class='big' width='120' align='center'>
<font size='3' color='000066'>
A Few Clouds<br>
<br>71&deg;F<br>(22&deg;C)</td>
</font><td rowspan='2' width='200'><table cellspacing='0' cellpadding='2' border='0'
width='100%'>
<tr bgcolor='#b0c4de'>
<td><b>Humidity</b>:</td>
<td align='right'>30 %</td>
</tr>
<tr bgcolor='#ffefd5'>
<td><b>Wind Speed</b>:</td><td align='right'>SE 5 MPH<br>
</td>
</tr>
<tr bgcolor='#b0c4de'>
<td><b>Barometer</b>:</td><td align='right' nowrap>30.05 in (1015.90 mb)</td></tr>
<tr bgcolor='#ffefd5'>
<td><b>Dewpoint</b>:</td><td align='right'>38&deg;F (3&deg;C)</td>
</tr>
</tr>
<tr bgcolor='#ffefd5'>
<td><b>Visibility</b>:</td><td align='right'>10.00 Miles</td>
</tr>
<tr><td nowrap><b><a
href='http://www.wrh.noaa.gov/total_forecast/other_obs.php?wfo=psr&zone=AZZ023'
class='link'>More Local Wx:</a></b> </td>
<td nowrap align='right'><b><a
href='http://www.wrh.noaa.gov/mesowest/getobext.php?wfo=psr&sid=KPHX&num=72' class='link'>3
Day History:</a></b> </td></tr>
</table>
...
```

There are two approaches to narrowing your attention to the specific area of the document in which you are interested. You can either search the HTML for a word or phrase close to the data that you want, or, as we mentioned previously, use Google Chrome or Firefox with Firebug to "Inspect Element" and see the element you want embedded in an attractive diagram of the document tree. Figure 10–2 shows Google Chrome with its Developer Tools pane open following an Inspect Element command: my mouse is poised over the `<font>` element that was brought up in its document tree, and the element itself is highlighted in blue on the web page itself.

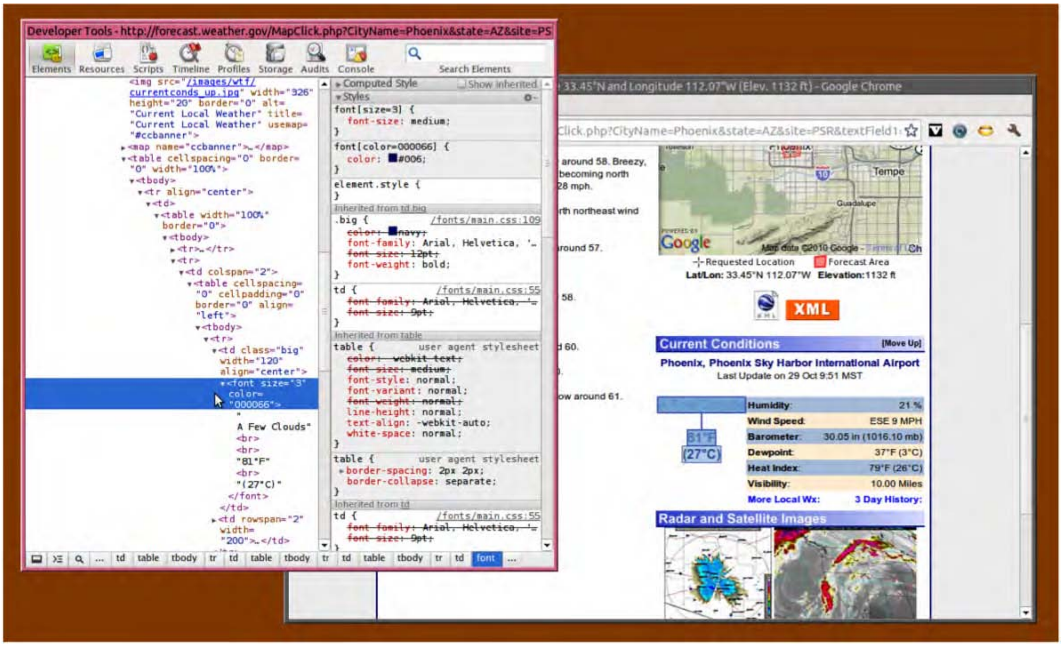


Figure 10–2. Examining Document Elements in the Browser

Note that Google Chrome does have an annoying habit of filling in "conceptual" tags that are not actually present in the source code, like the `<tbody>` tags that you can see in every one of the tables shown here. For that reason, I look at the actual HTML source before writing my Python code; I mainly use Chrome to help me find the right places in the HTML.

We will want to grab the text "A Few Clouds" as well as the temperature before turning our attention to the table that sits to this element's right, which contains the humidity.

A properly indented version of the HTML page that you are scraping is good to have at your elbow while writing code. I have included `phoenix-tidied.html` with the source code bundle for this chapter so that you can take a look at how much easier it is to read!

You can see that the element displaying the current conditions in Phoenix sits very deep within the document hierarchy. Deep nesting is a very common feature of complicated page designs, and that is why simply walking a document object model can be a very verbose way to select part of a document—and, of course, a brittle one, because it will be sensitive to changes in any of the target element's parent. This will break your screen-scraping program not only if the target web site does a redesign, but also simply because changes in the time of day or the need for the site to host different kinds of ads can change the layout subtly and ruin your selector logic.

To see how direct document-object manipulation would work in this case, we can load the raw page directly into both the `lxml` and `BeautifulSoup` systems.

```
>>> import lxml.etree
>>> parser = lxml.etree.HTMLParser(encoding='utf-8')
>>> tree = lxml.etree.parse('phoenix.html', parser)
```

The need for a separate parser object here is because, as you might guess from its name, `lxml` is natively targeted at XML files.

```
>>> from BeautifulSoup import BeautifulSoup
>>> soup = BeautifulSoup(open('phoenix.html'))
Traceback (most recent call last):
  ...
HTMLParseError: malformed start tag, at line 96, column 720
```

What on earth? Well, look, the National Weather Service does not check or tidy its HTML! I might have chosen a different example for this book if I had known, but since this is a good illustration of the way the real world works, let's press on. Jumping to line 96, column 720 of `phoenix.html`, we see that there does indeed appear to be some broken HTML:

```
<a href="http://www.weather.gov"<u>www.weather.gov</u></a>
```

You can see that the `<u>` tag starts before a closing angle bracket has been encountered for the `<a>` tag. But why should BeautifulSoup care? I wonder what version I have installed.

```
>>> BeautifulSoup.__version__
'3.1.0'
```

Well, drat. I typed too quickly and was not careful to specify a working version when I ran `pip` to install BeautifulSoup into my virtual environment. Let's try again:

```
$ pip install BeautifulSoup==3.0.8.1
```

And now the broken document parses successfully:

```
>>> from BeautifulSoup import BeautifulSoup
>>> soup = BeautifulSoup(open('phoenix.html'))
```

That is much better!

Now, if we were to take the approach of starting at the top of the document and digging ever deeper until we find the node that we are interested in, we are going to have to generate some very verbose code. Here is the approach we would have to take with `lxml`:

```
>>> fonttag = tree.find('body').find('div').findall('table')[3] \
...     .findall('tr')[1].find('td').findall('table')[1].find('tr') \
...     .findall('td')[1].findall('table')[1].find('tr').find('td') \
...     .find('table').findall('tr')[1].find('td').find('table') \
...     .find('tr').find('td').find('font')
>>> fonttag.text
'\nA Few Clouds'
```

An attractive syntactic convention lets BeautifulSoup handle some of these steps more beautifully:

```
>>> fonttag = soup.body.div('table', recursive=False)[3] \
...     ('tr', recursive=False)[1].td('table', recursive=False)[1].tr \
...     ('td', recursive=False)[1]('table', recursive=False)[1].tr.td \
...     .table('tr', recursive=False)[1].td.table \
...     .tr.td.font
>>> fonttag.text
u'A Few Clouds71&deg;F(22&deg;C)'
```

BeautifulSoup lets you choose the first child element with a given tag by simply selecting the attribute `.tagname`, and lets you receive a list of child elements with a given tag name by calling an element like a function—you can also explicitly call the method `findAll()`—with the tag name and a `recursive` option telling it to pay attention just to the children of an element; by default, this option is set to `True`, and BeautifulSoup will run off and find all elements with that tag in the entire sub-tree beneath an element!

Anyway, two lessons should be evident from the foregoing exploration.

First, both `lxml` and BeautifulSoup provide attractive ways to quickly grab a child element based on its tag name and position in the document.

Second, we clearly should not be using such primitive navigation to try descending into a real-world web page! I have no idea how code like the expressions just shown can easily be debugged or maintained; they would probably have to be re-built from the ground up if anything went wrong with them—they are a painful example of write-once code.

And that is why selectors that each screen-scraping library supports are so critically important: they are how you can ignore the many layers of elements that might surround a particular target, and dive right in to the piece of information you need.

Figuring out how HTML elements are grouped, by the way, is much easier if you either view HTML with an editor that prints it as a tree, or if you run it through a tool like HTML tidy from W3C that can indent each tag to show you which ones are inside which other ones:

```
$ tidy phoenix.html > phoenix-tidied.html
```

You can also use either of these libraries to try tidying the code, with a call like one of these:

```
lxml.html.tostring(html)
soup.prettify()
```

See each library's documentation for more details on using these calls.

Selectors

A selector is a pattern that is crafted to match document elements on which your program wants to operate. There are several popular flavors of selector, and we will look at each of them as possible techniques for finding the current-conditions `<font>` tag in the National Weather Service page for Phoenix. We will look at three:

- People who are deeply XML-centric prefer XPath expressions, which are a companion technology to XML itself and let you match elements based on their ancestors, their own identity, and textual matches against their attributes and text content. They are very powerful as well as quite general.
- If you are a web developer, then you probably link to CSS selectors as the most natural choice for examining HTML. These are the same patterns used in Cascading Style Sheets documents to describe the set of elements to which each set of styles should be applied.
- Both `lxml` and BeautifulSoup, as we have seen, provide a smattering of their own methods for finding document elements.

Here are standards and descriptions for each of the selector styles just described— first, XPath:

```
http://www.w3.org/TR/xpath/
http://codespeak.net/lxml/tutorial.html#using-xpath-to-find-text
http://codespeak.net/lxml/xpathxslt.html
```

And here are some CSS selector resources:

```
http://www.w3.org/TR/CSS2/selector.html
http://codespeak.net/lxml/cssselect.html
```

And, finally, here are links to documentation that looks at selector methods peculiar to lxml and BeautifulSoup:

```
http://codespeak.net/lxml/tutorial.html#elementpath
http://www.crummy.com/software/BeautifulSoup/documentation.html#Searching the Parse Tree
```

The National Weather Service has not been kind to us in constructing this web page. The area that contains the current conditions seems to be constructed entirely of generic untagged elements; none of them have id or class values like currentConditions or temperature that might help guide us to them.

Well, what are the features of the elements that contain the current weather conditions in Listing 10–3? The first thing I notice is that the enclosing <td> element has the class "big". Looking at the page visually, I see that nothing else seems to be of exactly that font size; could it be so simple as to search the document for every <td> with this CSS class? Let us try, using a CSS selector to begin with:

```
>>> from lxml.cssselect import CSSSelector
>>> sel = CSSSelector('td.big')
>>> sel(tree)
[<Element td at b72ec0a4>]
```

Perfect! It is also easy to grab elements with a particular class attribute using the peculiar syntax of BeautifulSoup:

```
>>> soup.find('td', 'big')
<td class="big" width="120" align="center">
<font size="3" color="000066">
A Few Clouds<br />
<br />71&deg;F<br />(22&deg;C)</font></td>
```

Writing an XPath selector that can find CSS classes is a bit difficult since the class="" attribute contains space-separated values and we do not know, in general, whether the class will be listed first, last, or in the middle.

```
>>> tree.xpath(".//td[contains(concat(' ', normalize-space(@class), ' '), ' big ')]")
[<Element td at a567fcc>]
```

This is a common trick when using XPath against HTML: by prepending and appending spaces to the class attribute, the selector assures that it can look for the target class name with spaces around it and find a match regardless of where in the list of classes the name falls.

Selectors, then, can make it simple, elegant, and also quite fast to find elements deep within a document that interest us. And if they break because the document is redesigned or because of a corner case we did not anticipate, they tend to break in obvious ways, unlike the tedious and deep procedure of walking the document tree that we attempted first.

Once you have zeroed in on the part of the document that interests you, it is generally a very simple matter to use the ElementTree or the old BeautifulSoup API to get the text or attribute values you need. Compare the following code to the actual tree shown in Listing 10–3:

```
>>> td = sel(tree)[0]
>>> td.find('font').text
'\nA Few Clouds'
>>> td.find('font').findall('br')[1].tail
u'71°F'
```

If you are annoyed that the first string did not return as a Unicode object, you will have to blame the ElementTree standard; the glitch has been corrected in Python 3! Note that ElementTree thinks of text strings in an HTML file not as entities of their own, but as either the `.text` of its parent element or the `.tail` of the previous element. This can take a bit of getting used to, and works like this:

```
<p>
  My favorite play is    # the <p> element's .text
  <i>
»   Hamlet                 # the <i> element's .text
  </i>
  which is not really      # the <i> element's .tail
  <b>
»   Danish                 # the <b> element's .text
  </b>
  but English.             # the <b> element's .tail
</p>
```

This can be confusing because you would think of the three words `favorite` and `really` and `English` as being at the same "level" of the document—as all being children of the `<p>` element somehow—but `lxml` considers only the first word to be part of the text attached to the `<p>` element, and considers the other two to belong to the tail texts of the inner `<i>` and `<b>` elements. This arrangement can require a bit of contortion if you ever want to move elements without disturbing the text around them, but leads to rather clean code otherwise, if the programmer can keep a clear picture of it in her mind.

BeautifulSoup, by contrast, considers the snippets of text and the `<br>` elements inside the `<font>` tag to all be children sitting at the same level of its hierarchy. Strings of text, in other words, are treated as phantom elements. This means that we can simply grab our text snippets by choosing the right child nodes:

```
>>> td = soup.find('td', 'big')
>>> td.font.contents[0]
u'\nA Few Clouds'
>>> td.font.contents[4]
u'71&deg;F'
```

Through a similar operation, we can direct either `lxml` or BeautifulSoup to the humidity datum. Since the word `Humidity:` will always occur literally in the document next to the numeric value, this search can be driven by a meaningful term rather than by something as random as the `big` CSS tag. See Listing 10–4 for a complete screen-scraping routine that does the same operation first with `lxml` and then with BeautifulSoup.

This complete program, which hits the National Weather Service web page for each request, takes the city name on the command line:

```
$ python weather.py Springfield, IL
Condition:
Traceback (most recent call last):
  ...
AttributeError: 'NoneType' object has no attribute 'text'
```

And here you can see, superbly illustrated, why screen scraping is always an approach of last resort and should always be avoided if you can possibly get your hands on the data some other way: because presentation markup is typically designed for one thing—human readability in browsers—and can vary in crazy ways depending on what it is displaying.

What is the problem here? A short investigation suggests that the NWS page includes only a `<font>` element inside of the `<tr>` if—and this is just a guess of mine, based on a few examples—the description of the current conditions is several words long and thus happens to contain a space. The conditions in Phoenix as I have written this chapter are "A Few Clouds," so the foregoing code has worked just fine;

but in Springfield, the weather is "Fair" and therefore does not need a `<font>` wrapper around it, apparently.

Listing 10–4. *Completed Weather Scraper*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 10 - weather.py
# Fetch the weather forecast from the National Weather Service.

import sys, urllib, urllib2
import lxml.etree
from lxml.cssselect import CSSSelector
from BeautifulSoup import BeautifulSoup

if len(sys.argv) < 2:
»   print >>sys.stderr, 'usage: weather.py CITY, STATE'
»   exit(2)

data = urllib.urlencode({'inputstring': ' '.join(sys.argv[1:])})
info = urllib2.urlopen('http://forecast.weather.gov/zipcity.php', data)
content = info.read()

# Solution #1
parser = lxml.etree.HTMLParser(encoding='utf-8')
tree = lxml.etree.fromstring(content, parser)
big = CSSSelector('td.big')(tree)[0]
if big.find('font') is not None:
»   big = big.find('font')
print 'Condition:', big.text.strip()
print 'Temperature:', big.findall('br')[1].tail
tr = tree.xpath('.//td[b="Humidity"]')[0].getparent()
print 'Humidity:', tr.findall('td')[1].text
print

# Solution #2
soup = BeautifulSoup(content)  # doctest: +SKIP
big = soup.find('td', 'big')
if big.font is not None:
»   big = big.font
print 'Condition:', big.contents[0].string.strip()
temp = big.contents[3].string or big.contents[4].string  # can be either
print 'Temperature:', temp.replace('&deg;', ' ')
tr = soup.find('b', text='Humidity').parent.parent.parent
print 'Humidity:', tr('td')[1].string
print
```

If you look at the final form of Listing 10–4, you will see a few other tweaks that I made as I noticed changes in format with different cities. It now seems to work against a reasonable selection of locations; again, note that it gives the same report twice, generated once with `lxml` and once with BeautifulSoup:

```
$ python weather.py Springfield, IL
Condition: Fair
Temperature: 54 °F
Humidity: 28 %
```

```
Condition: Fair
Temperature: 54  F
Humidity: 28 %

$ python weather.py Grand Canyon, AZ
Condition: Fair
Temperature: 67°F
Humidity: 28 %

Condition: Fair
Temperature: 67 F
Humidity: 28 %
```

You will note that some cities have spaces between the temperature and the `F`, and others do not. No, I have no idea why. But if you were to parse these values to compare them, you would have to learn every possible variant and your parser would have to take them into account.

I leave it as an exercise to the reader to determine why the web page currently displays the word "NULL"—you can even see it in the browser—for the temperature in Elk City, Oklahoma. Maybe that location is too forlorn to even deserve a reading? In any case, it is yet another special case that you would have to treat sanely if you were actually trying to repackage this HTML page for access from an API:

```
$ python weather.py Elk City, OK
Condition: Fair and Breezy
Temperature: NULL
Humidity: NA

Condition: Fair and Breezy
Temperature: NULL
Humidity: NA
```

I also leave as an exercise to the reader the task of parsing the error page that comes up if a city cannot be found, or if the Weather Service finds it ambiguous and prints a list of more specific choices!

Summary

Although the Python Standard Library has several modules related to SGML and, more specifically, to HTML parsing, there are two premier screen-scraping technologies in use today: the fast and powerful `lxml` library that supports the standard Python "ElementTree" API for accessing trees of elements, and the quirky BeautifulSoup library that has powerful API conventions all its own for querying and traversing a document.

If you use BeautifulSoup before 3.2 comes out, be sure to download the most recent 3.0 version; the 3.1 series, which unfortunately will install by default, is broken and chokes easily on HTML glitches.

Screen scraping is, at bottom, a complete mess. Web pages vary in unpredictable ways even if you are browsing just one kind of object on the site—like cities at the National Weather Service, for example.

To prepare to screen scrape, download a copy of the page, and use HTML tidy, or else your screen-scraping library of choice, to create a copy of the file that your eyes can more easily read. Always run your program against the ugly original copy, however, lest HTML tidy fixes something in the markup that your program will need to repair!

Once you find the data you want in the web page, look around at the nearby elements for tags, classes, and text that are unique to that spot on the screen. Then, construct a Python command using your scraping library that looks for the pattern you have discovered and retrieves the element in question. By looking at its children, parents, or enclosed text, you should be able to pull out the data that you need from the web page intact.

When you have a basic script working, continue testing it; you will probably find many edge cases that have to be handled correctly before it becomes generally useful. Remember: when possible, always use true APIs, and treat screen scraping as a technique of last resort!

CHAPTER 11

Web Applications

This chapter focuses on the actual act of programming—on what it means to sit down and write a Python web application. Every other issue that we consider will be in the service of this overarching goal: to create a new web service using Python as our language.

The work of designing a web site can be enormous and incur months of graphic design and usability work. Or it can involve nothing more than a single-page sketch on the back of a napkin. It can even be as simple as an idea in your head. But when it comes to implementation, applications that are designed to face the public Internet demand at least three big decisions from their implementers:

- A front-end web server will need to be chosen. Its job is to listen on port 80 of the web server—or whatever port has been designated for the site—and to serve static content like images, style sheets, and JavaScript files. And, for the specific URLs that serve the actual dynamic site content, the front-end server needs to delegate page creation to your Python program.
- Some means of linking the server and the Python application needs to be selected. We will spend the most time on the WSGI standard, which provides a standard invocation protocol between a web server and Python; however, it is also common for servers and Python to be linked through mechanisms like FastCGI and SCGI.
- Either in the web server itself or in the harness that runs the Python code, there needs to be logic that spawns several copies of the Python web application code, whether as threads or processes. This enables your app to answer different customers simultaneously without blocking.
- Finally, the programmer needs to decide which Python libraries he will use for common tasks like URL dispatch, database access, and template rendering—or whether to do without the convenience of standard tools altogether and to roll some of these solutions on his own. Often he will choose to use a web framework that provides these features as a more-or-less unified suite.

Very often, the process of building a web application goes through these bullet points in reverse order. Most often, a programmer starts experimenting with an idea by running the "create project" routine of a popular web framework and adding her own code to the skeleton that gets created. Days or weeks or months later, when it is time to start exposing her application to real users on the local intranet or even out on the World Wide Web, the developer belatedly researches the best choice of front-end server for her framework of choice. She spends a few hours getting everything tweaked and configured correctly, so she can put her application into production.

But we will tackle the steps in the order listed previously, moving from the front end of the system towards its core. This means that we will first establish the context in which Python web services run, and then spend the rest of the chapter focusing on actual programming techniques.

Web Servers and Python

Acceptable web site performance generally requires the ability to serve several users concurrently. And since few Python programmers condescend to writing their web application logic using Twisted callbacks (see Chapter 7), achieving this performance means running several copies of your web application concurrently, using either threads or processes.

You will recall from our discussion of threads in Chapter 7 that the standard C language implementation of Python—the version of Python people download from its web site—does not actually run Python code in a thread-safe manner. To avoid corrupting in-memory data structures, C Python employs a Global Interpreter Lock (GIL), so that only one thread in a multi-threaded program can actually be executing Python code at any given time. Thus Python will let you create as many threads as you want in a given process; however, only one thread can run code at a time, as though your threads were confined to a single processor.

You might think that multiprocessing would always be the required approach; however, it turns out that threading can have decent performance because so many web applications are essentially light front-ends that sit between the user and a database. A typical web application receives and parses the user's request, then makes a corresponding request to the database behind it; while that thread is waiting for a response from the database, the GIL is available for any other threads that need to run Python code. Finally the database answers; the waiting thread reacquires the GIL; and, in a quick blaze of CPU activity, the data is turned into an attractive web page, and the response is sent winging its way back to the user.

Thus threads can sometimes at least perform decently. Nevertheless, multiple processes are the more general way to scale. This is because, as a service gets bigger, additional processes can be brought up on additional machines, rather than being confined to a single machine. Threads, no matter their other merits, cannot do that!

There are two general approaches to running a Python web application inside of a collector of identical worker processes:

- The Apache web server can be combined with the popular `mod_wsgi` module to host a separate Python interpreter in every Apache worker process.
- The web application can be run inside of either the `flup` server or the `uWSGI` server. Both of these servers will manage a pool of worker processes where each process hosts a Python interpreter running your application. The front-end web server can submit requests to `flup` using either the standard Fast CGI (FCGI) or Simple CGI (SCGI) protocol, while it has to speak to `uWSGI` in its own special "uwsgi" protocol (whose name is all lowercase to distinguish it from the name of the server).

Note that both approaches insist that a powerful, secure, name-brand web server face the actual customer, with your Python web application sitting safely behind it. This lets the web server use its fast, compiled code to reject obviously malformed or nonsensical HTTP requests, passing along to your application only those requests that are at least superficially parsable. It can also have performance benefits, as you will see in the next section.

Two Tiers

Most objects fetched by your web browser each day are completely static: images, movies, style sheets, and JavaScript files. These are usually served directly from disk, without undergoing any dynamic modification or customization. Modern web pages with even fairly simple designs typically include a dozen or more static elements for every dynamically-generated page you actually visit—elements that will remain the same for weeks or months until, in fact, the site is upgraded or redesigned.

This will be familiar to you if you have ever used the Google Chrome built-in Developer Tools or the Firebug extension for Firefox to look behind the scenes and see the resources that must be downloaded to actually display a web page. For example, Figure 11–1 shows the files that Google Chrome downloads to display the Stack Overflow front page. The time axis goes from left to right, with the entire download taking about 1.5 seconds. The last few outliers appear to be advertisements, which are often the slowest elements of a web site to load.

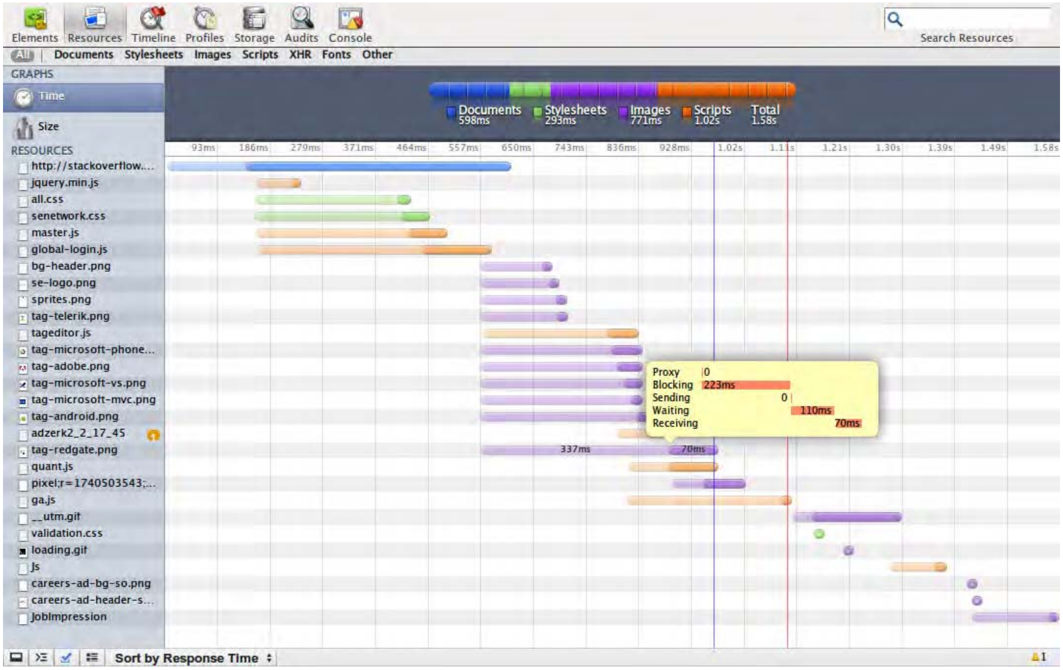


Figure 11–1. Downloading the Stack Overflow Front Page

For our purposes, the most important fact in this graph is that, of all of the many resources that make up the front page, it is likely that only one of them—the initial HTML page itself, whose download is displayed as the upper-left bar on the graph—was generated dynamically. The HTML contains the dynamic list of most-active questions, along with user-specific information such as my name and the list of tags that I find interesting. Everything else is completely generic; all the images, styles, and scripts remain exactly the same, regardless of who is visiting the site.

And so production web apps are best designed with two tiers of servers:

- The first server actually faces users and their browsers. It should be configured to serve the high-volume static content directly from disk using a fast, static language like C.
- The second server is a framework that powers the dynamic pages. It is invoked only for pages that absolutely require it. Often the dynamic code runs in a separate process that listens on a `localhost` port that only the front-end web server can access. (see Chapters 2 and 3 for more about sockets and `localhost`.)

Many administrators are tempted to run only one web server that combines these two roles. They accomplish this by choosing a very flexible front-end web server that can also directly host their application code. But having two separate servers for the static and dynamic content has a number of benefits, including the ability to performance tune the servers separately. For example, the front-end workers can be small and light to answer requests for static content, while the back-end worker processes can be fewer but heavier because they each need to host a full Python interpreter. If you try running just one server, then every worker will need to contain both the lightweight code for serving static files and the Python interpreter for creating dynamic pages, but only one or the other piece of code will get invoked for a given request.

Choosing a Web Server

All of the popular open source web servers can be used to serve Python web applications, so the full range of modern options is available:

> *Apache HTTP Server:* Since taking the lead as the most popular HTTP server back in 1996, Apache has always remained in the top spot and has never yet been eclipsed by a competitor. Its stated goal is flexibility and modularity; it is reasonably fast, but it will not win speed records against more recent servers that focus only on speed. Its configuration files can be a bit long and verbose, but through them Apache offers very powerful options for applying different rules and behaviors to different directories and URLs. A variety of extension modules are available (many of which come bundled with it), and user directories can have separate `.htaccess` configuration files that make further adjustments to the main configuration.
>
> *nginx ("engine X"):* Started by a Russian programmer in the early 2000's, the `nginx` server has become a great favorite of organizations with a large volume of content that needs to be served quickly. It is considered fairly easy to configure.
>
> *lighttpd ("lighty"):* First written to demonstrate an architecture that could support tens of thousands of open client sockets (both `nginx` and Cherokee are also contenders in this class), this server is known for being very easy to configure. Some system administrators complain about its memory usage, but many others have observed no problems with it.
>
> *Cherokee:* Not only does this server offer performance that might edge out even `nginx` and `lighttpd`, but it lets you configure the server through a built-in web interface.

Of course, this list will grow slowly out-of-date over time, so you should use it only as a jumping-off point for your own research into choosing an HTTP server. Nevertheless, having a good list of specific examples at this point is important because it enables us to turn to the concrete question of Python integration.

So how can each of these servers be combined with Python?

One option, of course, is to simply set up Apache and configure it to serve all of your content, both static and dynamic.

Alternatively, the `mod_wsgi` module has a *daemon* mode where it internally runs your Python code inside a stack of dedicated server processes that are separate from Apache. Each WSGI process can even run as a different user. If you really want to use Apache as your front end, this is one of the best options available.

But the most strongly recommended approach today is to set up one of the three fast servers to provide your static content, and then use one of the following three techniques to run your Python code behind them:

- Use HTTP proxying so that your nginx, lighttpd, or Cherokee front-end server delivers HTTP requests for dynamic web pages to a back-end Apache instance running `mod_wsgi`.
- Use the FastCGI protocol or SCGI protocol to talk to a `flup` instance running your Python code.
- Use the uwsgi protocol to talk to a `uWSGI` instance running your Python code.

Given that every one of the four major web servers supports HTTP, the fast CGI protocols, and uwsgi, your options are quite broad. So how do you decide on a specific approach?

Your first task should be to look at the documentation, tweets, and blogs for the Python web framework or tools on which you intend to build your solution. Choosing a configuration that is a standard in that community increases your chances of success; it also increases the possibility of getting useful help if things go wrong.

Also, list any specific features that you require of your front end and choose only from among the HTTP servers that support them. Make sure your choice can support your requirements involving certificates and encryption, as well as any restrictions you want placed on SSL protocol versions or permitted ciphers (see Chapter 6). You should also make sure your choice runs well on the operating system you will be deploying. If your operating system vendor (like Red Hat or Ubuntu) already provides precompiled versions of any of these servers, then that might also deserve consideration.

As mentioned previously, the task of selecting and configuring a front-end web server often comes quite late in the timeline of a project; and the choice will draw much more deeply upon your system administrator skills than it will upon your expertise as a programmer.

At this point, you understand something of the larger context in which Python web applications are usually run; you are now ready to turn your attention to the task of programming.

WSGI

When the front-end web server receives an HTTP request, consults the patterns listed in its configuration, and decides that this particular URL needs to be handled by a Python web application, how does it actually invoke the Python code? And how can that code then communicate back to the server, whether to signal an error, make a redirect, or return a particular block of data as the web page?

Integrating Python with web servers used to be the Wild West: every server presented programmers with different data formats and calling conventions. Small web programs written against one server's API would need to be ported before they could be used with another brand of web server; and web frameworks themselves had to maintain a separate entry point for each server which developers might want to use to deploy their applications.

This situation was much improved by the creation of PEP 333, which defines the Python Web Server Gateway Interface (WSGI): `www.python.org/dev/peps/pep-0333/`

WSGI introduced a single calling convention that every web server could implement, thereby making that web server instantly compatible with all of the Python web applications and web frameworks that also support WSGI.

Developers generally avoid writing raw WSGI applications because the conveniences of even a simple web framework make code so much easier to write and maintain. But, for the sake of illustration, Listing 10-1 shows a small WSGI application whose front page asks the user to type a string. Submitting the string takes the user to a second web page, where he can see its base64 encoding. From there, a link will take him back to the first page to repeat the process.

Listing 11–1. *A Complete WSGI Application*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 11 - wsgi_app.py
# A simple web application built directly against the low-level WSGI spec.

import cgi, base64
from wsgiref.simple_server import make_server

def page(content, *args):
    yield '<html><head><title>wsgi_app.py</title></head><body>'
    yield content % args
    yield '</body>'

def simple_app(environ, start_response):
    gohome = '<br><a href="/">Return to the home page</a>'
    q = cgi.parse_qs(environ['QUERY_STRING'])

    if environ['PATH_INFO'] == '/':

        if environ['REQUEST_METHOD'] != 'GET' or environ['QUERY_STRING']:
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: the front page is not a form']

        start_response('200 OK', [('Content-Type', 'text/html')])
        return page('Welcome! Enter a string: <form action="encode">'
                    '<input name="mystring"><input type="submit"></form>')

    elif environ['PATH_INFO'] == '/encode':

        if environ['REQUEST_METHOD'] != 'GET':
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: this form does not support POST parameters']

        if 'mystring' not in q or not q['mystring'][0]:
            start_response('400 Bad Request', [('Content-Type', 'text/plain')])
            return ['Error: this form requires a "mystring" parameter']

        my = q['mystring'][0]
        start_response('200 OK', [('Content-Type', 'text/html')])
        return page('<tt>%s</tt> base64 encoded is: <tt>%s</tt>' + gohome,
                    cgi.escape(repr(my)), cgi.escape(base64.b64encode(my)))

    else:
        start_response('404 Not Found', [('Content-Type', 'text/plain')])
        return ['That URL is not valid']

print 'Listening on localhost:8000'
make_server('localhost', 8000, simple_app).serve_forever()
```

The first thing to note in this code listing is that two very different objects are being created: a WSGI server that knows how to use HTTP to talk to a web browser and an application written to respond correctly when invoked per the WSGI calling convention. Note that these two pieces—the client and server—could easily be swapped out. Other WSGI applications would all work equally well when run by

the wsgiref simple server; similarly, any other web server that speaks WSGI could be substituted in place of the wsgiref server and thus be used to serve this particular application.

This code example should make the calling convention clear enough:

1. For each incoming request, the application is called with an environ object, giving it the details of the HTTP request and a live, callable, and named start_response().
2. Once the application has decided what HTTP response code and headers need to be returned, it makes a single call to start_response(). Its headers will be combined with any headers that the WSGI server might already provide to the client.
3. Finally, the application needs only to return the actual content—either a list of strings or a generator yielding strings. Either way, the strings will be concatenated by the WSGI server to produce the response body that is transmitted back to the client. Generators are useful for cases where it would be unwise for an application to try loading all of the content (like large files) into memory at once.

Of course, using this apparently simple convention in the real world involves all sorts of caveats and edge cases, and a leisurely read through PEP 333 should satisfy any further curiosity you have about what counts as appropriate behavior in WSGI applications and servers.

I have tried to make this tiny application formally correct in order to illustrate the onus that WSGI places upon the programmer to handle every possible situation. After checking the URL against the paths to the two pages that exist on this small site, it correctly returns a 404 error, complete with helpful error text. For each page that does exist, any confused attempts on the part of an HTTP client to submit inappropriate data need to return an error (which is generally friendlier than leaving a user confused as to why her input is not making any difference). Content types have to be paired correctly with documents; and, of course, user input has to be meticulously quoted so that any special HTML characters are never copied literally into any document that we return.

Because programmers do not generally enjoy solving these problems over and over again in every application they write, very few Python programmers tend to write raw WSGI applications. Instead, they use web frameworks that provide tools for handling URLs (replacing the big if statement in Listing 11–1); for interpolating user strings into HTML and other markup; and for handling non-existent URLs and badly written forms automatically—all without explicitly writing clauses to detect such circumstances in every application.

Note that the wsgiref package, whose simple_server we used here, also contains several utilities for working with WSGI. It includes functions for examining, further unpacking, and modifying the environ object; a prebuilt iterator for streaming large files back to the server; and even a validate sub-module whose routines can check a WSGI application to see whether it complies with the specification when presented with a series of representative requests.

WSGI Middleware

Standard interfaces like WSGI make it possible for developers to create *wrappers*—a design-patterns person would call these *adapters*—that accept a request from a server; modify, adjust, or record the request; and then call a normal WSGI application with the modified environment. Such middleware can also inspect and adjust the outgoing data stream; everything, in fact, is up for grabs, and essential arbitrary changes can be made both to the circumstances under which a WSGI application runs, as well as to the content that it returns. Many other possibilities leap to mind, like these (in cases where I just mention a bare module name, you can visit its Python Package Index page to learn more about it):

- If several WSGI applications need to live at a single web site under different URLs, then a piece of middleware can be given the URLs. This middleware can then delegate incoming requests to the correct application, returning 404 errors on its own authority when requests arrive where the URL doesn't match any of the configuration applications. Ian Bicking wrote Paste Deploy (you can learn more at `pythonpaste.org`), a tool that combines exactly this kind of URL routing with a simple system for centrally managing the configuration of several WSGI applications.
- If each WSGI application on a web site were to keep its own list of passwords and honor only its own session cookies, then users would have to log in again each time they crossed an application boundary. By delegating authentication to WSGI middleware, applications can be relieved even of the duty to provide their own login page; instead, the middleware asks a user who lacks a session cookie to log in; once a user is authenticated, the middleware can pass along the user's identity to the applications by putting the user's information in the `environ` argument. Both repoze.who and repoze.what can help site integrators assert site-wide control over users and their permissions.
- Theming can be a problem when several small applications are combined to form a larger web site. This is because each application typically has its own approach to theming. One usually has to learn as many new theming systems as there are applications to combine! This has led to the development of two competing tools, xdv and Deliverance, that let you build a single HTML theme and then provide simple rules that pull text out of your back-end applications and drop it into your theme in the right places.
- Debuggers can be created that call a WSGI application and, if an uncaught Python exception is raised, display an annotated traceback to support debugging. WebError actually provides the developer with a live, in-browser Python command line prompt for every level in a stack trace at which the developer can investigate a failure. Another popular tool is repoze.profile, which watches the application as it processes requests and produces a report on which functions are consuming the most CPU cycles.

If you are interested in what WSGI middleware is available, then you can visit this pair of sites to learn more:

```
http://wsgi.org/wsgi/Middleware_and_Utilities
http://repoze.org/repoze_components.html#middleware
```

Over the next few years, new ideas will continue to emerge while old solutions start to fade into obscurity. Therefore, be sure to check current blogs, mailing lists, and Stack Overflow when looking for middleware solutions.

Having said all of that, I think it is fair to observe that most Python web programmers today are not making very active use of WSGI middleware.

The “weak” version of the WSGI gospel has certainly come to pass: all Python web frameworks seem to support WSGI. But the “strong” version of the WSGI gospel has been slower to arrive: WSGI has not become the standard mechanism by which Python web applications are constructed. That said, a few Python web programmers certainly exist who tend to build sites by taking a few small WSGI applications and placing them behind a stack of middleware that handles URL dispatch, authentication, and theming.

Today there are at least three major competing approaches in the Python community for crafting modular components that can be used to build web sites:

- The WSGI middleware approach thinks that code reuse can often best be achieved through a component stack, where each component uses WSGI to speak to the next. Here, all interaction has to somehow be made to fit the model of a dictionary of strings being handed down and then content being passed back up.
- Everything built atop the Zope Toolkit uses formal Design Pattern concepts like interfaces and factories to let components discover one another and be configured for operation. Thanks to adapters, components can often be used with widgets that were not originally designed with a given type of component in mind.
- Several web frameworks have tried to adopt conventions that would make it easy for third-party pieces of functionality to be added to an application easily. The Django community seems to have traveled the farthest in this direction, but it also looks as though it has encountered quite serious roadblocks in cases where a component needs to add its own tables to the database that have foreign-key relationships with user tables.

These examples illustrate an important fact: WSGI middleware is a good idea that has worked very well for a small class of problems where the idea of wrapping an application with concentric functionality makes solid sense. However, most web programmers seem to want to use more typical Python mechanisms like APIs, classes, and objects to combine their own code with existing components. The problem is that we, as a Python community, are still learning about different ways of accomplishing this, and no single “Pythonic” solution appears ready to emerge.

One of the biggest changes in Python 3 is its much more rigorous approach toward byte strings and Unicode strings. These can be mixed so freely in Python 2 that developers often use the wrong type without knowing it. Obviously, the transition to Python 3 will heavily affect WSGI because strings and their encodings are a foundational issue both in parsing HTTP requests and in generating well-formed responses. PEP 444 is currently the focus of this work. Pay attention to this PEP's status as you contemplate moving your web applications to Python 3; it should point you to any further resources you will need to understand how web application stacks will communicate in the future.

Python Web Frameworks

And here, in the middle of this book on Python network programming, we reach what for many of you will be the jumping off point into an entirely different discipline: web application development.

Network programmers think about things like sockets, port numbers, protocols, packet loss, latency, framing, and encodings. Although all of these concepts must also be in the back of a web developer's mind, her actual attention is focused on a set of technologies so intricate and fast-changing that the actual packets and latencies are recalled to mind only when they are causing trouble. The web developer needs to think instead about HTML, `GET`, `POST`, forms, REST, CSS, JavaScript, Ajax, APIs, sprites, compression, and emerging technologies like HTML5 and WebSocket. The web site exists in her mind primarily as a series of documents that users will traverse to accomplish goals.

Web frameworks exist to help programmers step back from the details of HTTP—which is, after all, an implementation detail most users never even become aware of—and to write code that focuses on the nouns of web design. Listing 11–2 shows how even a very modest Python microframework can be used to reorient the attention of a web programmer. You can install the framework and run the listing once you have activated a virtual environment (see Chapter 1):

```
$ pip install bottle
$ python bottle_app.py
Bottle server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Use Ctrl-C to quit.
```

Listing 11–2 also requires an accompanying template file, which is shown in Listing 11–3.

Listing 11–2. *Rewriting the WSGI Application With a Framework*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 11 - wsgi_app.py
# A simple web application built using the Bottle micro-framework.

import base64, bottle
bottle.debug(True)
app = bottle.Bottle()

@app.route('/encode')
@bottle.view('bottle_template.html')
def encode():
»   mystring = bottle.request.GET.get('mystring')
»   if mystring is None:
»   »   bottle.abort(400, 'This form requires a "mystring" parameter')
»   return dict(mystring=mystring, myb=base64.b64encode(mystring))

@app.route('/')
@bottle.view('bottle_template.html')
def index():
»   return dict(mystring=None)

bottle.run(app=app, host='localhost', port=8080)
```

In Listing 11–1, the attention was on the single incoming HTTP request, and the branches in our logic explored all of the possible lifespans for that particular protocol request. Listing 11–2 changes the focus to the pages that actually exist on the site and giving each of these pages reasonable behaviors. The same tree of possibilities exists, but the tree exists implicitly thanks to the possible URLs defined in the code, not because the programmer has written a large `if` statement.

Listing 11–3. *The Template That Goes With Listing 11–2*

```
%#!/usr/bin/env python
%# Foundations of Python Network Programming - Chapter 11 - bottle_template.py
%# The page template that goes with bottle_app.py.
%#
<html><head><title>bottle_app.py</title></head>
<body>
  %if mystring is None:
»   Welcome! Enter a string:
»   <form action="encode"><input name="mystring"><input type="submit"></form>
  %else:
»   <tt>{{mystring}}</tt> base64 encoded is: <tt>{{myb}}</tt><br>
»   <a href="/">Return to the home page</a>
  %end
</body>
```

It might seem merely a pleasant convenience that we can use the Bottle `SimpleTemplate` to insert our variables into a web page and know that they will be escaped correctly. But the truth is that templates serve, just like schemes for URL dispatch, to re-orient our attention: instead of the resulting web page existing in our minds as what will result when the strings in our program listing are finally concatenated, we get to lay out its HTML intact, in order, and in a file that can actually take an `.html` extension and be

highlighted and indented as HTML in our editor. The Python program will no longer impede our relationship with our markup.

And full-fledged Python frameworks abstract away even more implementation details. A very important feature they typically provide is data abstraction: instead of talking to a database using its raw APIs, a programmer can define *models*, laying out the data fields so they are easy to instantiate, search, and modify. And some frameworks can provide entire RESTful APIs that allow creation, inspection, modification, and deletion with `PUT`, `GET`, `POST`, and `DELETE`. The programmer merely needs to define the structure of his data document, and then name the URL at which the tree of REST objects should be based.

So how should you go about choosing a web framework?

It might surprise you, but I am not going to now launch into a comparative review of my favorite web frameworks. I think that the field is simply moving too fast. Old web frameworks lose steam, new ones appear, and the ones with the best communities keep innovating and have the annoying habit of making complaints about missing features suddenly obsolete.

So, while I will mention a few good web frameworks by name in the text that follows, you should probably start your search for a web framework at the Python Wiki page, where the community keeps a list of the available contenders: `http://wiki.python.org/moin/WebFrameworks`

Next—and I cannot emphasize this enough—you should always be participating in a programming community. Find a local Python meet-up or users group, or fellow students or employees who are using Python to solve problems similar to yours. Ask them what web frameworks they are using. Using the second-best Python web framework in the world—which will still be a pretty good one—in the company of other people who are also using it, blogging about it, and contributing patches to it will contribute to a vastly more wonderful experience that sitting alone in the dark and using the best framework instead.

Online community is important too. Once you have chosen a web framework, be sure to check out its mailing lists, forums, IRC channels, and its important blogs. You should also look for answers to questions you might have on Stack Overflow. Stay connected, and you will often save days or weeks of effort when helpful fellow Python programmers point you at better solutions for your problems than the ones you knew existed.

When looking for a web framework, you will find that the various frameworks differ on a few major points. The upcoming sections will walk you through what these points are, and how they might affect your development experience.

URL Dispatch Techniques

The various Python web frameworks tend to handle URL dispatch quite differently.

- Some small frameworks like Bottle and Flask let you create small applications by decorating a series of callables with URL patterns; small applications can then be combined later by placing them beneath one or more top-level applications.

- Others frameworks, like Django, Pylons, and Werkzeug, encourage each application to define its URLs all in one place. This breaks your code into two levels, where URL dispatch happens in one location and rendering in another. This separation makes it easier to review all of the URLs that an application supports; it also means that you can attach code to new URLs without having to modify the functions themselves.

- Another approach has you define controllers, which are classes that represent some point in the URL hierarchy—say, the path `/cart`—and then write methods on the controller class named `view()` and `edit()` if you want to support sub-pages named `/cart/view` and `/cart/edit`. CherryPy, TurboGears2, and Pylons (if you use controllers instead of Routes) all support this approach. While determining later what URLs are supported can mean traversing a maze of different connected classes, this approach does allow for dynamic, recursive URL spaces that exist only at runtime as classes hand off dispatch requests based on live data about the site structure.
- A large community with its own conferences exists around the Zope framework. The Plone CMS is built atop Zope technology, and recent web frameworks like Grok and BFG have been springing up to try to make Zope more accessible. In this approach, URLs actually traverse your database! Instead of having to match string patterns or descend across a series of controllers, each URL is a path from one object to another, with each URL component naming the next object attribute that should be dereferenced. Zope people tend to store their objects in a Python object database like the ZODB, but traversal can work just as easily against objects stored in a relational database behind an ORM like SQLAlchemy. The last URL component, such as `/edit` or `/view`, usually selects one of several available views that are available to render the object.

When looking for a URL dispatch mechanism, pay particular attention to how each framework thinks about URLs. Recall from Chapter 9 that the following pair of URLs are different; the first has two path elements, while the second has three:

```
http://example.com/Nord%2FLB/logo
http://example.com/Nord/LB/logo
```

If you are building a fresh web application from the ground up and can absolutely guarantee that you will never need a slash in a URL path component—which you can assure by designing your application so that you always use web-ready slugs rather than raw item names, for example—then the distinction is not important. In that case, you can simply avoid ugly characters in path components entirely. But if you might need to support URLs like the first one listed in the preceding example, then beware that some popular Python web frameworks do not fully support RFC 1738 because they decode the `%2F` to a slash before your application ever sees it; this would make it impossible for you to properly distinguish literal slashes from encoded ones.

The various mechanisms for URL dispatch can all be used to produce fairly clean design, and choosing from among them is largely a matter of taste.

Templates

Almost all web frameworks expect you to produce web pages by combining Python code called a *view* with an HTML template; you saw this approach in action in Listing 11–2. This approach has gained traction because of its eminent maintainability: building a dictionary of information is best performed in plain Python code, and the items fetched and arranged by the view can then easily be included by the template, so long as the template language supports basic actions like iteration and some form of expression evaluation. It is one of the glories of Python that we use views and templates, and one of the shames of traditional PHP development that developers would freely intermix HTML and extensive PHP code to produce a single, unified mess.

Views can also become more testable when their only job is to generate a dictionary of data. A good framework will let you write tests that simply check the raw data returned by the function instead of making you peek repeatedly into fully rendered templates to see if the view corralled its data correctly.

There seem to be two major differences of opinion among the designers and users of the various template languages about what constitutes the best way to use templates:

- Should templates be valid HTML with iteration and expressions hidden in element attributes? Or should the template language use its own style of markup that festoons and wraps the literal HTML of the web page, as in Listing 11–3? While the former can let the developer run HTML validation against template files before they are ever rendered and be assured that rendering will not change the validator's verdict, most developers seem to find the latter approach much easier to read and maintain.

- Should templates allow arbitrary Python expressions in template code, or lock down the available options to primitive operations like dictionary get-item and object get-attribute? Many popular frameworks choose the latter option, requiring even lazy programmers to push complex operations into their Python code "where it belongs." But several template languages reason that, if Python programmers do so well without type checking, then maybe they should also be trusted with the choice of which expressions belong in the view and which in the template.

Since many Python frameworks let you plug in your template language of choice, and only a few of them lock you down to one option, you might find that you can pair your favorite approaches. As always, look for a community with a consistent practice, try to understand the reasons for the community's choice of template language, and join that community if you think its approach holds water.

Final Considerations

A few last thoughts on web frameworks will conclude our discussion.

Because Python is powerful and flexible, it is an easy language in which to write new web frameworks. These frameworks can even spring up accidentally, as an application that originally needed to serve "just one tiny status web page" gradually grows its own custom URL dispatch mechanism, template conventions, and view calling conventions. But while web frameworks are easy to create, you should generally avoid creating new ones. Instead, it is very often best to simply choose an existing framework. You will benefit from all of the work and knowledge about HTTP that has gone into its design, and other programmers will be able to understand your code if they are already familiar with the framework you have chosen.

The Django web framework wins many converts because its *admin interface* makes it easy to browse the back-end database. Each table row gets rendered as the corresponding Django model object that the developer has defined, and an administrator can use simple forms to create, edit, or delete objects. This not only lets data-entry personnel get to work immediately on populating a model while web developers are still getting started on designing the application itself, but it also means that developers often never have to write CRUD (create, read, update, and delete) pages for objects that are not manipulated by end-users.

You might think that a database browser would serve just as well, but over time a web application tends to develop a lot of knowledge about its models that is never pushed back into the database. For example, a web application knows which fields can have which combinations of values and which string formats are allowed and disallowed. Only by editing a database row with the application logic itself can these invariants and constraints be enforced. Other web frameworks have tried to add a system like the Django admin interface, but none of these alternatives seem to have succeeded to the degree that Django has at the time of writing.

Web services, whether RPC or RESTful, are another important feature that are simple to write in some frameworks but are rather more verbose or difficult in others. Most large web applications today contain many dynamic elements—like URLs that are not human-browsable, but which return data for

the sake of JavaScript routines running in each web page. If your application will require these, then check each framework that interests you for its degree of support for XML-RPC, JSON, and a RESTful approach to data documents in general.

Obviously, user authentication is an enormous issue for many programmers. Some web frameworks remember usernames and passwords and feature support for login screens and session cookies out-of-the-box; others make you build these components yourself or plug in a third-party approach to solve these issues. Still others have such complex systems of extensions and plug-ins that some developers can never get user configuration working in the desired manner.

If you need specific advanced features such as a Comet-like approach to serving events back to the browser, then you should obviously look for a framework that implements those required features well. Note that some quite successful web applications are produced by combining two or more frameworks that are loosely coupled through a common database. For example, you might combine a Django application serving HTML pages with a `restish` application that provides RESTful CRUD operations that can be invoked from JavaScript. Choosing the best tool for the job sometimes involves combining several tools, instead of demanding everything from a single monolithic application.

Finally, understand that experience is often the best guide to choosing a web framework. This means that, after all of the window shopping, you are going to have to sit down and try actually writing in some of the various frameworks available before you really know whether a given framework will prove a good fit for either yourself or your problem. We all like to think that we know ourselves well enough to predict such things without doing any actual coding. For example, we may feel we know intuitively which web framework is really “our style” and will let us be productive. However, trying a range of approaches may reveal that your initial guess about how your mind works did not actually do justice to what you can accomplish with an ostensibly “ugly” framework, once you understand its benefits.

Remember that, whatever framework you choose, you are writing Python code. This means that your application's quality is going to depend at least as much upon your ability to write good, clean, Pythonic code as it will upon any magical features of the framework itself. The framework might even feel rather awkward to you, or it might be selected by a senior team member with whom you disagree; but always remember that this is Python, and that its flexibility and elegance will generally let you write a good web application, whatever framework you ultimately end up using.

Pure-Python Web Servers

A fun way to demonstrate that Python comes with “batteries included” is to enter a directory on your system and run the `SimpleHTTPServer` Standard Library module as a stand-alone program:

```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

If you direct your browser to `localhost:8000`, you will see the contents of this script's current directory displayed for browsing, such as the listings provided by Apache when a site leaves a directory browsable. Documents and images will load in your web browser when selected, based on the content types chosen through the best guesses of the `mimetypes` Standard Library module.

The `SimpleHTTPServer` is a subclass of `BaseHTTPServer`, which is also the foundation of the `wsgiref.simple_server` that we looked at earlier. Before the WSGI standard was invented, small Python programs could subclass `BaseHTTPServer` if they needed to answer raw HTTP requests. This is actually a common difference between the old mechanisms by which Python programs provided extensibility, and the more modern and Pythonic mechanisms. It used to be popular to write a class with stub methods inside, and then tell programmers to extend it by subclassing and defining those methods; today, we use namespaces, callables, and duck-typed objects to provide much cleaner forms of extensibility. For example, today an object like `start_response` is provided as an argument (dependency injection), and

the WSGI standard specifies its behavior rather than its inheritance tree (duck typing).The Standard Library includes two other HTTP servers:

- `CGIHTTPServer` takes the `SimpleHTTPServer` and, instead of just serving static files off of the disk, it adds the ability to run CGI scripts (which we will cover in the next section).
- `SimpleXMLRPCServer` and `DocXMLRPCServer` each provide a server endpoint against which client programs can make XML-RPC remote procedure calls, as demonstrated in Chapter 18 and Listing 22-1. This protocol uses XML files submitted through HTTP requests.

Note that none of the preceding servers is typically intended for production use; instead, they are useful for small internal tasks for which you just need a quick HTTP endpoint to be used by other services internal to a system or subnet. And while most Python web frameworks will provide a way to run your application from the command line for debugging, most follow the lead of Django in recommending against using the development platform for debugging.

However, a few Python web frameworks exist that not only provide their own built-in HTTP servers, but have also worked on their security and performance, so that they can be recommended for use in production.

These pure-Python web servers can be very useful if you are writing an application that users will be installing locally, and you want to provide a web interface without having to ship a separate web server like Apache or nginx. Frameworks that offer a pure-Python integrated web server include CherryPy; Zope, and thus Grok and BFG; web2py; and web.py.

CGI

When the first experiments were taking place with dynamically generated web pages, developers would write an external program and have their web server run the program every time a matching HTTP request arrived. This resembled the behavior of the traditional `inetd` server (Chapter 7), which could run an external command to answer each incoming TCP or UDP connection on a listening socket.

Many of these early web servers lacked the ability to designate entire sections of a web site as dynamic; for example, you could not tell them, “all URLs beneath `/cart` should be handled by my application.” Instead, scripts were simply designated as a new type of content, and then placed right next to static pages, images, and archive files in directories that were already browsable from the web. Just as servers had been told that `.html` files should be delivered as `text/html` content and that `.jpg` files should be returned as images, they were now told that executable `.cgi` files should be run and their output returned to the HTTP client.

Obviously, a calling convention was necessary, and so the Common Gateway Interface (CGI) was defined. It allowed programs in all sorts of languages—C, the various Unix shells, awk, Perl, Python, PHP, and so forth—to be partners in generating dynamic content.

Today, the design of CGI is considered something of a disaster. Running a new process from scratch is just about the most expensive single operation that you can perform on a modern operating system, and requiring that this take place for every single incoming HTTP request is simply madness. You should avoid CGI under all circumstances. But it is possible you might someday have to connect Python code to a legacy HTTP server that does not support at least FastCGI or SCGI, so I will outline CGI's essential features.

Three standard lines of communication that already existed between parent and child processes on Unix systems were used by web servers when invoking a CGI script:

- The Unix *environment*—a list of strings provided to each process upon its invocation that traditionally includes things like `TZ=EST` (the time zone) and `COLUMNS=80` (user's screen width)—was instead stuffed full of information about the HTTP request that the CGI script was being called upon to answer. The various parts of the request's URL; the user agent string; basic information about the web server; and even a cookie could be included in the list of colon-separated key-value pairs.

- The standard input to the script could be read to end-of-file to receive whatever data had been submitted in the body of the HTTP request using `POST`. Whether a request was indeed a `POST` could be checked by examining the `REQUEST_METHOD` environment variable.

- Finally, the script would produce content, which it did by writing HTTP headers, a blank line, and then a response body to its standard output. To be a valid response, a `Content-Type` header was generally necessary at a minimum—though in its absence, some web servers would instead accept a `Location` header as a signal that they should send a redirect.

Should you ever need to run Python behind an HTTP server that only supports CGI, then I recommend that you use the `CGIHandler` module from the `wsgiref` Standard Library package. This lets you use a normal Python web framework to write your service—or, alternatively, to roll up your sleeves and write a raw WSGI application—and then offer the HTTP server a CGI script, as shown here:

```
import CGIHandler, MyWSGIApp
my_wsgi_app = MyWSGIApp()  # configuration necessary here?
CGIHandler().run(my_wsgi_app)
```

Be sure to check whether your web framework of choice already provides a way to invoke it as a CGI script; if so, your web framework will already know all of the steps involved in loading and configuring your application. For complex web frameworks, it might be tedious to run all of the steps manually in a script like this one.

The Python Standard Library also includes two more ancient modules related to the CGI protocol.

Old CGI scripts written with Python imported the `cgi` module; this module contains a number of helpers and utilities for interpreting both the standard CGI environment variables and also for parsing `GET` or `POST` form parameters, so that you can access them like a dictionary.

And, believe it or not, there is actually a `CGIHTTPServer` module in the Standard Library, so that Python can actually be used as a CGI-enabled HTTP server. Like many other web servers of its era, this module interprets URLs as paths into a directory tree and serves static files directly while invoking CGI scripts as separate processes.

mod_python

As it became clear that CGI was both inefficient and inflexible—CGI scripts could not flexibly set the HTTP return code, for example—it became fashionable to start embedding programming languages directly in web servers.

Earlier in this chapter, we discussed how embedding Python is possible today with `mod_wsgi` under Apache. (We also noted that this is usually only desirable if you have another web server in front of Apache that can handle static requests with workers that are not bloated by including their own Python interpreter!) Back in the early days, embedding was also possible, through a somewhat different approach that actually made Python an extension language for much of the internals of Apache itself. The module that supported this was `mod_python`, and for years it was by far the most popular way to connect Python to the World Wide Web.

The mod_python Apache module put a Python interpreter inside of every worker process spawned by Apache. Programmers could arrange for their Python code to be invoked by writing directives into their Apache configuration files like this:

```
<Directory /cart>
  AddHandler mod_python .py
  PythonHandler my_shopping_cart
  PythonDebug On
</Directory>
```

All kinds of Apache handlers could be provided, each of which intervened at a different moment during the various stages of Apache's request processing. Most Python programmers just declared a *publisher* handler—*publishing* was one of Apache's last steps and the one where content was generated. These scripts would examine their request argument and then build a response, which made them look like a more complex version of this snippet:

```
# "my_shopping_cart.py"
from mod_python import apache
def handler(request):
»   request.content_type = 'text/plain'
»   request.write('Welcome to your Python-powered shopping cart!')
»   return apache.OK
```

But many other kinds of handlers could also be implemented, thanks to the many integration points that mod_python provided. Python could be used to make access control decisions, to authenticate requests using non-standard mechanisms, to pre-process headers before they were handed off to some other application, and to implement custom logging or statistics collection.

It must be admitted, though, that very little of the excitement surrounding this flexibility ever seems to have panned out. I think that lots of complex logic that people once dreamed of plugging into Apache actually wound up being implemented inside web frameworks instead. The frameworks tended to take things like authentication and redirection out of the web server altogether. But for the few people who really did need to extend Apache, nothing could really replace mod_python.

Today, mod_python is mainly of historical interest. I have outlined its features here, not only because you might be called upon to maintain or upgrade a service that is still running on mod_python, but because it still provides unique Apache integration points where Python cannot get involved in any other way. If you run into either situation, you can find its documentation at modpython.org.

Summary

Web applications are typically deployed by using either a pure-Python web server for a low-volume or internal site, or by using a high-capacity front-end server to serve static content and dispatch requests for dynamic pages to your Python application. A popular approach is to put something fast like nginx, lighttpd, or Cherokee in front, and then use flup, uWSGI, or Apache with mod_wsgi to actually manage your Python server processes.

The introduction of the WSGI calling convention in PEP 333 has been a great advance in Python web interoperability: web servers and web applications can now be paired freely, freeing web framework developers from having to build in explicit support for every web server they wanted to support. Each WSGI application is a callable that receives information about the incoming web request and issues an HTTP response code, headers, and content in reply.

WSGI middleware is software that sits between a WSGI server and an application, and performs operations like authentication, dispatch, tracing, and debugging. Middleware is especially useful when several applications are being combined and themed to form a single web site. However, it is fair to say that, at this point, the dream has not come to pass that Python web developers would one day start new

web applications by selecting and configuring a middleware stack that got the application's boilerplate logic out of the way.

Python web frameworks are crucial to modern web development. They handle much of the logic of HTTP, and they also provide several important abstractions: they can dispatch different URLs to different Python code, insert Python variables into HTML templates, and provide important assistance in both persisting Python objects to the database and also in letting them be accessed from the web both through user-facing CRUD interfaces as well as RESTful web-service protocols.

There do exist pure-Python web servers, which can be especially important when writing a web interface for a program that users will install locally. There are not only good choices available for download, but a few small servers are even built into the Python Standard Library.

Two old approaches to dynamic web page generation are the CGI protocol and the `mod_python` Apache module. Neither should be used for new development.

CHAPTER 12

E-mail Composition and Decoding

The early e-mail protocols were among the first network dialects developed for the Internet. The world was a simple one in those days: everyone with access to the Internet reached it through a command-line account on an Internet-connected machine. There, at the command line, they would type out e-mails to their friends, and then they could check their in-boxes when new mail arrived. The entire task of an e-mail protocol was to transmit messages from one big Internet server to another, whenever someone sent mail to a friend whose shell account happened to be on a different machine.

Today the situation is much more complicated: not only is the network involved in moving e-mail between servers, but it is often also the tool with which people check and send e-mail. I am not talking merely about webmail services, like Google Mail; those are really just the modern versions of the command-line shell accounts of yesteryear, because the mail that Google's web service displays in your browser is still being stored on one of Google's big servers. Instead, a more complicated situation arises when someone uses an e-mail client like Mozilla Thunderbird or Microsoft Outlook that, unlike Gmail, is running locally on their desktop or laptop.

In this case of a local e-mail client, the network is involved in three different ways as a message is transmitted and received:

- First, the e-mail client program submits the message to a server on the Internet on which the sender has an e-mail account. This usually takes place over Authenticated SMTP, which we will learn about in Chapter 13.
- Next, that e-mail server finds and connects to the server named as the destination of the e-mail message —the server in charge of the domain named after the `@` sign. This conversation takes place over normal, vanilla, un-authenticated SMTP. Again, Chapter 13 is where you should go for details.
- Finally, the recipient uses Thunderbird or Outlook to connect to his or her e-mail server and discover that someone has sent a new message. This could take place over any of several protocols—probably over an older protocol called POP, which we cover in Chapter 14, but perhaps over the modern IMAP protocol to which we dedicate Chapter 15.

You will note that all of these e-mail protocols are discussed in the subsequent chapters of this book. What, then, is the purpose of this chapter? Here, we will learn about the actual payload that is carried by all of the aforementioned protocols: the format of e-mail messages themselves.

E-mail Messages

We will start by looking at how old-fashioned, plain-text e-mail messages work, of the kind that were first sent on the ancient Internet. Then, we will learn about the innovations and extensions to this format that today let e-mail messages support sophisticated formats, like HTML, and that let them include attachments that might contain images or other binary data.

■ **Caution** The `email` module described in this chapter has improved several times through its history, making leaps forward in Python versions 2.2.2, 2.4, and 2.5. Like the rest of this book, this chapter focuses on Python 2.5 and later. If you need to use older versions of the `email` module, first read this chapter, and then consult the Standard Library documentation for the older version of Python that you are using to see the ways in which its `email` module differed from the modern one described here.

Each traditional e-mail message contains two distinct parts: headers and the body. Here is a very simple e-mail message so that you can see what the two sections look like:

```
From: Jane Smith <jsmith@example.com>
To: Alan Jones <ajones@example.com>
Subject: Testing This E-Mail Thing

Hello Alan,
This is just a test message. Thanks.
```

The first section is called the headers, which contain all of the metadata about the message, like the sender, the destination, and the subject of the message —everything except the text of the message itself. The body then follows and contains the message text itself.

There are three basic rules of Internet e-mail formatting:

- At least during actual transmission, every line of an e-mail message should be terminated by the two-character sequence carriage return, newline, represented in Python by `'\r\n'`. E-mail clients running on your laptop or desktop machine tend to make different decisions about whether to store messages in this format, or replace these two-character line endings with whatever ending is native to your operating system.
- The first few lines of an e-mail are headers, which consist of a header name, a colon, a space, and a value. A header can be several lines long by indenting the second and following lines from the left margin as a signal that they belong to the header above them.
- The headers end with a blank line (that is, by two line endings back-to-back without intervening text) and then the message body is everything else that follows. The body is also sometimes called the payload.

The preceding example shows only a very minimal set of headers, like a message might contain when an e-mail client first sends it. However, as soon as it is sent, the mail server will likely add a `Date` header, a `Received` header, and possibly many more. Most mail readers do not display all the headers of

a message, but if you look in your mail reader's menus for an option like as "show all headers" or "view source," you should be able to see them.

Take a look at Listing 12–1 to see a real e-mail message from a few years ago, with all of its headers intact.

Listing 12–1. *A Real-Life E-mail Message*

```
Delivered-To: brandon@europa.gtri.gatech.edu
Received: from pele.santafe.edu (pele.santafe.edu [192.12.12.119])
        by europa.gtri.gatech.edu (Postfix) with ESMTP id 6C4774809
        for <brandon@rhodesmill.org>; Fri,  3 Dec 1999 04:00:58 -0500 (EST)
Received: from aztec.santafe.edu (aztec [192.12.12.49])
        by pele.santafe.edu (8.9.1/8.9.1) with ESMTP id CAA27250
        for <brandon@rhodesmill.org>; Fri, 3 Dec 1999 02:00:57 -0700 (MST)
Received: (from rms@localhost)
        by aztec.santafe.edu (8.9.1b+Sun/8.9.1) id CAA29939;
Fri, 3 Dec 1999 02:00:56 -0700 (MST)
Date: Fri, 3 Dec 1999 02:00:56 -0700 (MST)
Message-Id: <199912030900.CAA29939@aztec.santafe.edu>
X-Authentication-Warning: aztec.santafe.edu: rms set sender to rms@gnu.org
        using -f
From: Richard Stallman <rms@gnu.org>
To: brandon@rhodesmill.org
In-reply-to: <m3k8my7x1k.fsf@europa.gtri.gatech.edu> (message from Brandon
        Craig Rhodes on 02 Dec 1999 00:04:55 -0500)
Subject: Re: Please proofread this license
Reply-To: rms@gnu.org
References: <199911280547.WAA21685@aztec.santafe.edu>
        <m3k8my7x1k.fsf@europa.gtri.gatech.edu>
Xref: 38-74.clients.speedfactory.net scrapbook:11
Lines: 1

Thanks.
```

Yes, those are a lot of headers for a mere one-line thank-you message! It is, in fact, common for the headers of short e-mail messages to overwhelm the actual size of the message itself.

There are many more headers here than in the first example. Let's take a look at them.

First, notice the `Received` headers. These are inserted by mail servers. Each mail server through which the message passes adds a new `Received` header, above the others —so you should read them in the final message from bottom to top. You can see that this message passed through four mail servers.

Some mail server along the way —or possibly the mail reader —added the `Sender` line, which is similar to the `From` line. The `Mime-Version` and `Content-Type` headers will be discussed later on in this chapter, in the "Understanding MIME" section. The `Message-ID` header is supposed to be a globally unique way to identify any particular message, and is generated by either the mail reader or mail server when the message is first sent. The `Lines` header indicates the length of the message. Finally, the mail reader that I used at the time, Gnus, added an `X-Mailer` header to advertise its involvement in composing the message. (This can help server administrators in debugging when an e-mail arrives with a formatting problem, letting them trace the cause to a particular e-mail program.)

If you viewed this message in a normal mail reader, you would likely see only `To`, `From`, `Subject`, and `Date` by default. The Internet e-mail standard is extremely stable; even though this message is several years old, it would still be perfectly valid today.

As we will learn in the following chapters, the headers of an e-mail message are not actually part of routing the message to its recipients; the SMTP protocol receives a list of destination addresses for each message that is kept separate from the actual headers and text of the message itself. The headers are there for the benefit of the person who reads the e-mail message, and the most important headers are these:

- `From`: This identifies the message sender. It can also, in the absence of a `Reply-to` header, be used as the destination when the reader clicks the e-mail client's "Reply" button.
- `Reply-To`: This sets an alternative address for replies, in case they should go to someone besides the sender named in the `From` header.
- `Subject`: This is a short several-word description of the e-mail's purpose, used by most clients when displaying whole mailboxes full of e-mail messages.
- `Date`: This is a header that can be used to sort a mailbox in the order in which e-mails arrived.
- `Message-ID` and `In-Reply-To`: Each ID uniquely identifies a message, and these IDs are then used in e-mail replies to specify exactly which message was being replied to. This can help sophisticated mail readers perform "threading," arranging messages so that replies are grouped directly beneath the messages to which they reply.

There are also a whole set of MIME headers, which help the mail reader display the message in the proper language, with proper formatting, and which help e-mail clients process attachments correctly; we will learn more about them shortly.

Composing Traditional Messages

Now that you know what a traditional e-mail looks like, how can we generate one in Python without having to implement the formatting details ourselves? The answer is to use the modules within the powerful `email` package.

As our first example, Listing 12–2 shows a program that generates a simple message. Note that when you generate messages this way, manually setting the payload with the `Message` class, you should limit yourself to using plain 7-bit ASCII text.

Listing 12–2. *Creating an E-mail Message*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - trad_gen_simple.py
# Traditional Message Generation, Simple
# This program requires Python 2.5 or above

from email.message import Message
text = """Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous"""

msg = Message()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 12'
msg.set_payload(text)

print msg.as_string()
```

The program is simple. It creates a `Message` object, sets the headers and body, and prints the result. When you run this program, you will get a nice formatted message with proper headers. The output is suitable for transmission right away! You can see the result in Listing 12–3.

Listing 12–3. *Printing the E-mail to the Screen*

```
$ ./trad_gen_simple.py
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 12

Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous
```

While technically correct, this message is actually a bit deficient when it comes to providing enough headers to really function in the modern world. For one thing, most e-mails should have a `Date` header, in a format specific to e-mail messages. Python provides an `email.utils.formatdate()` routine that will generate dates in the right format.

You should add a `Message-ID` header to messages. This header should be generated in such a way that no other e-mail, anywhere in history, will ever have the same Message-ID. This might sound difficult, but Python provides a function to help do that as well: `email.utils.make_msgid()`.

So take a look at Listing 12–4, which fleshes out our first sample program into a more complete example that sets these additional headers.

Listing 12–4. *Generating a More Complete Set of Headers*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - trad_gen_newhdrs.py
# Traditional Message Generation with Date and Message-ID
# This program requires Python 2.5 or above

import email.utils
from email.message import Message

message = """Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous"""

msg = Message()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 12'
msg['Date'] = email.utils.formatdate(localtime = 1)
msg['Message-ID'] = email.utils.make_msgid()
msg.set_payload(message)

print msg.as_string()
```

That's better! If you run the program, you will notice two new headers in the output, as shown in Listing 12–5.

Listing 12–5. *A More Complete E-mail Is Printed Out*

```
$ ./trad_gen_newhdrs.py
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 12
Date: Mon, 02 Aug 2010 10:05:55 -0400
Message-ID: <20100802140555.11734.89229@guinness.ten22>

Hello,

This is a test message from Chapter 12.  I hope you enjoy it!
-- Anonymous
```

The message is now ready to send!

You might be curious how the unique `Message-ID` is created. It is generated by adhering to a set of loose guidelines. The part to the right of the `@` is the full hostname of the machine that is generating the e-mail message; this helps prevent the message ID from being the same as the IDs generated on entirely different computers. The part on the left is typically generated using a combination of the date, time, the process ID of the program generating the message, and some random data. This combination of data tends to work well in practice in making sure every message can be uniquely identified.

Parsing Traditional Messages

So those are the basics of creating a plain e-mail message. But what happens when you receive an incoming message as a raw block of text and want to look inside? Well, the `email` module also provides support for parsing e-mail messages, re-constructing the same `Message` object that would have been used to create the message in the first place. (Of course, it does not matter whether the e-mail you are parsing was originally created in Python through the `Message` class, or whether some other e-mail program created it; the format is standard, so Python's parsing should work either way.)

After parsing the message, you can easily access individual headers and the body of the message using the same conventions as you used to create messages: headers look like the dictionary key-values of the `Message`, and the body can be fetched with a function. A simple example of a parser is shown in Listing 12–6. All of the actual parsing takes place in the one-line function `message_from_file()`; everything else in the program listing is simply an illustration of how a `Message` object can be mined for headers and data.

Listing 12–6. *Parsing and Displaying a Simple E-mail*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - trad_parse.py
# Traditional Message Parsing
# This program requires Python 2.5 or above

import email

banner = '-' * 48
popular_headers = ('From', 'To', 'Subject', 'Date')
msg = email.message_from_file(open('message.txt'))
headers = sorted(msg.keys())

print banner
```

```
for header in headers:
»   if header not in popular_headers:
»   »   print header + ':', msg[header]
print banner
for header in headers:
»   if header in popular_headers:
»   »   print header + ':', msg[header]
print banner
if msg.is_multipart():
»   print "This program cannot handle MIME multipart messages."
else:
»   print msg.get_payload()
```

Like many e-mail clients, this parser distinguishes between the few e-mail headers that users are actually likely to want visible —like `From` and `Subject`—and the passel of additional headers that are less likely to interest them. If you save the e-mail shown in Listing 12–5 as `message.txt`, for example, then running `trad_parse.py` will result in the output shown in Listing 12–7.

Listing 12–7. *The Output of Our E-mail Parser*

```
$ ./trad_parse.py
------------------------------------------------
Message-ID: <20100802140555.11734.89229@guinness.ten22>
------------------------------------------------
Date: Mon, 02 Aug 2010 10:05:55 -0400
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 12
To: recipient@example.com
------------------------------------------------
Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous
```

Here, the "unpopular" `Message-ID` header, which most users just want hidden, is shown first. Then, the headers actually of interest to the user are printed. Finally, the body of the e-mail message is displayed on the screen.

As you can see, the Python Standard Library makes it quite easy both to create and then to parse standard Internet e-mail messages! Note that the `email` package also offers a `message_from_string()` function that, instead of taking a file, can simply be handed the string containing an e-mail message.

Parsing Dates

The `email` package provides two functions that work together as a team to help you parse the `Date` field of e-mail messages, whose format you can see in the preceding example: a date and time, followed by a time zone expressed as hours and minutes (two digits each) relative to UTC. Countries in the eastern hemisphere experience sunrise early, so their time zones are expressed as positive numbers, like the following:

```
Date: Sun, 27 May 2007 11:34:43 +1000
```

Those of us in the western hemisphere have to wait longer for the sun to rise, so our time zones lag behind; Eastern Daylight Time, for example, runs four hours behind UTC:

```
Date: Sun, 27 May 2007 08:36:37 -0400
```

Although the `email.utils` module provides a bare `parsedate()` function that will extract the components of the date in the usual Python order (starting with the year and going down through smaller increments of time), this is normally not what you want, because it omits the time zone, which you need to consider if you want dates that you can really compare (because, for example, you want to display e-mail messages in order they were written!).

To figure out what moment of time is really meant by a `Date` header, simply call two functions in a row:

- Call `parsedate_tz()` to extract the time and time zone.
- Use `mktime_tz()` to add or subtract the time zone.
- The result with be a standard Unix timestamp.

For example, consider the two `Date` headers shown previously. If you just compared their bare times, the first date looks later: 11:34 a.m. is, after all, after 8:36 a.m. But the second time is in fact the much later one, because it is expressed in a time zone that is so much farther west. We can test this by using the functions previously named. First, turn the top date into a timestamp:

```
>>> from email.utils import parsedate_tz, mktime_tz
>>> timetuple1 = parsedate_tz('Sun, 27 May 2007 11:34:43 +1000')
>>> print timetuple1
(2007, 5, 27, 11, 34, 43, 0, 1, -1, 36000)
>>> timestamp1 = mktime_tz(timetuple1)
>>> print timestamp1
1180229683.0
```

Then turn the second date into a timestamp as well, and the dates can be compared directly:

```
>>> timetuple2 = parsedate_tz('Sun, 27 May 2007 08:36:37 -0400')
>>> timestamp2 = mktime_tz(timetuple2)
>>> print timestamp2
1180269397.0
>>> timestamp1 < timestamp2
True
```

If you have never seen a timestamp value before, they represent time very plainly: as the number of seconds that have passed since the beginning of 1970. You will find functions in Python's old `time` module for doing calculations with timestamps, and you will also find that you can turn them into normal Python `datetime` objects quite easily:

```
>>> from datetime import datetime
>>> datetime.fromtimestamp(timestamp2)
datetime.datetime(2007, 5, 27, 8, 36, 37)
```

In the real world, many poorly written e-mail clients generate their `Date` headers incorrectly. While the routines previously shown do try to be flexible when confronted with a malformed `Date`, they sometimes can simply make no sense of it and `parsedate_tz()` has to give up and return `None`.

So when checking a real-world e-mail message for a date, remember to do it in three steps: first check whether a `Date` header is present at all; then be prepared for `None` to be returned when you parse it; and finally apply the time zone conversion to get a real timestamp that you can work with.

If you are writing an e-mail client, it is always worthwhile storing the time at which you first download or acquire each message, so that you can use that date as a substitute if it turns out that the message has a missing or broken `Date` header. It is also possible that the `Received:` headers that servers

have written to the top of the e-mail as it traveled would provide you with a usable date for presentation to the user.

Understanding MIME

So far we have discussed e-mail messages that are plain text: the characters after the blank line that ends the headers are to be presented literally to the user as the content of the e-mail message. Today, only a fraction of the messages sent across the Internet are so simple!

The Multipurpose Internet Mail Extensions (MIME) standard is a set of rules for encoding data, rather than simple plain text, inside e-mails. MIME provides a system for things like attachments, alternative message formats, and text that is stored in alternate encodings.

Because MIME messages have to be transmitted and delivered through many of the same old e-mail services that were originally designed to handle plain-text e-mails, MIME operates by adding headers to an e-mail message and then giving it content that looks like plain text to the machine but that can actually be decoded by an e-mail client into HTML, images, or attachments.

What are the most important features of MIME?

Well, first, MIME supports multipart messages. A normal e-mail message, as we have seen, contains some headers and a body. But a MIME message can squeeze several different parts into the message body. These parts might be things to be presented to the user in order, like a plain-text message, an image file attachment, and then a PDF attachment. Or, they could be alternative multiparts, which represent the same content in different ways —usually, by encoding a message in both plain text and HTML.

Second, MIME supports different transfer encodings. Traditional e-mail messages are limited to 7-bit data, which renders them unusable for international alphabets. MIME has several ways of transforming 8-bit data so it fits within the confines of e-mail systems:

- The “plain” encoding is the same as you would see in traditional messages, and passes 7-bit text unmodified.
- “Base-64” is a way of encoding raw binary data that turns it into normal alphanumeric data. Most of the attachments you send and receive —such as images, PDFs, and ZIP files —are encoded with base-64.
- “Quoted-printable” is a hybrid that tries to leave plain English text alone so that it remains readable in old mail readers, while also letting unusual characters be included as well. It is primarily used for languages such as German, which uses mostly the same Latin alphabet as English but adds a few other characters as well.

MIME also provides content types, which tell the recipient what kind of content is present. For instance, a content type of `text/plain` indicates a plain-text message, while `image/jpeg` is a JPEG image.

For text parts of a message, MIME can specify a character set. Although much of the computing world has now moved toward Unicode —and the popular UTF-8 encoding —as a common mechanism for transmitting international characters, many e-mail programs still prefer to choose a language-specific encoding. By specifying the encoding used, MIME makes sure that the binary codes in the e-mail get translated back into the correct characters on the user’s screen.

All of the foregoing mechanisms are very important and very powerful in the world of computer communication. In fact, MIME content types have become so successful that they are actually used by other protocols. For instance, HTTP uses MIME content types to state what kinds of documents it is sending over the Web.

How MIME Works

You will recall that MIME messages must work within the limited plain-text framework of traditional e-mail messages. To do that, the MIME specification defines some headers and some rules about formatting the body text.

For non-multipart messages that are a single block of data, MIME simply adds some headers to specify what kind of content the e-mail contains, along with its character set. But the body of the message is still a single piece, although it might be encoded with one of the schemes already described.

For multipart messages, things get trickier: MIME places a special marker in the e-mail body everywhere that it needs to separate one part from the next. Each part can then have its own limited set of headers —which occur at the start of the part —followed by data. By convention, the most basic content in an e-mail comes first (like a plain-text message, if one has been included), so that people without MIME-aware readers will see the plain text immediately without having to scroll down through dozens or hundreds of pages of MIME data.

Fortunately, Python knows all of the rules for generating and parsing MIME, and can support it all behind the scenes while letting you interact with an object-based representation of each message. Let us see how it works.

Composing MIME Attachments

We will start by looking at how to create MIME messages. To compose a message with attachments, you will generally follow these steps:

1. Create a `MIMEMultipart` object and set its message headers.
2. Create a `MIMEText` object with the message body text and attach it to the `MIMEMultipart` object.
3. Create appropriate MIME objects for each attachment and attach them to the `MIMEMultipart` object.
4. Finally, call `as_string()` on the `MIMEMultipart` object to write out the resulting message.

Take a look at Listing 12–8 for a program that implements this algorithm. You can see that parts of the code look similar to logic that we used to generate a traditional e-mail. After creating the message and its text body, the program loops over each file given on the command line and attaches it to the growing message. (If you run the program with an empty command line, then the message is simply printed without any attachments.)

Listing 12–8. *Creating a Simple MIME Message*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - mime_gen_basic.py
# This program requires Python 2.5 or above

from email.mime.base import MIMEBase
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email import utils, encoders
import mimetypes, sys

def attachment(filename):
»   fd = open(filename, 'rb')
```

```
    mimetype, mimeencoding = mimetypes.guess_type(filename)
    if mimeencoding or (mimetype is None):
        mimetype = 'application/octet-stream'
    maintype, subtype = mimetype.split('/')
    if maintype == 'text':
        retval = MIMEText(fd.read(), _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(fd.read())
        encoders.encode_base64(retval)
    retval.add_header('Content-Disposition', 'attachment',
            filename = filename)
    fd.close()
    return retval

message = """Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous"""
msg = MIMEMultipart()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 12'
msg['Date'] = utils.formatdate(localtime = 1)
msg['Message-ID'] = utils.make_msgid()

body = MIMEText(message, _subtype='plain')
msg.attach(body)
for filename in sys.argv[1:]:
    msg.attach(attachment(filename))
print msg.as_string()
```

The `attachment()` function does the work of creating a message attachment object. First, it determines the MIME type of each file by using Python's built-in `mimetypes` module. If the type can't be determined, or it will need a special kind of encoding, then a type is declared that promises only that the data is made of a "stream of octets" (sequence of bytes) but without any further promise about what they mean.

If the file is a text document whose MIME type starts with `text/`, a `MIMEText` object is created to handle it; otherwise, a `MIMEBase` generic object is created. In the latter case, the contents are assumed to be binary, so they are encoded with base-64. Finally, an appropriate `Content-Disposition` header is added to that section of the MIME file so that mail readers will know that they are dealing with an attachment.

The result of running this program is shown in Listing 12–9.

Listing 12–9. *Running the Program in Listing 12–8*

```
$ echo "This is a test" > test.txt
$ gzip < test.txt > test.txt.gz
$ ./mime_gen_basic.py test.txt test.txt.gz
Content-Type: multipart/mixed; boundary="===============1623374356=="
MIME-Version: 1.0
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 12
Date: Thu, 11 Dec 2003 16:00:55 -0600
```

```
Message-ID: <20031211220055.12211.26885@host.example.com>

--===============1623374356==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hello,
 This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous
--===============1623374356==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename="test.txt"

This is a test

--===============1623374356==
Content-Type: application/octet-stream
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="test.txt.gz"

H4sIAP3o2D8AAwvJyCxWAKJEhZLU4hIuAIwtwPoPAAAA
--===============1623374356==--
```

The message starts off looking quite similar to the traditional ones we created earlier; you can see familiar headers like `To`, `From`, and `Subject` just like before. Note the `Content-Type` line, however: it indicates `multipart/mixed`. That tells the mail reader that the body of the message contains multiple MIME parts, and that the string containing equals signs will be the separator between them.

Next comes the message's first part. Notice that it has its own `Content-Type` header! The second part looks similar to the first, but has an additional `Content-Disposition` header; this will signal most e-mail readers that the part should be displayed as a file that the user can save rather than being immediately displayed to the screen. Finally comes the part containing the binary file, encoded with base-64, which makes it not directly readable.

MIME Alternative Parts

MIME "alternative" parts let you generate multiple versions of a single document. The user's mail reader will then automatically decide which one to display, depending on which content type it likes best; some mail readers might even show the user radio buttons, or a menu, and let them choose.

The process of creating alternatives is similar to the process for attachments, and is illustrated in Listing 12–10.

Listing 12–10. *Writing a Message with Alternative Parts*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - mime_gen_alt.py
# This program requires Python 2.2.2 or above

from email.mime.base import MIMEBase
```

```
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email import utils, encoders

def alternative(data, contenttype):
»   maintype, subtype = contenttype.split('/')
»   if maintype == 'text':
»   »   retval = MIMEText(data, _subtype=subtype)
»   else:
»   »   retval = MIMEBase(maintype, subtype)
»   »   retval.set_payload(data)
»   »   encoders.encode_base64(retval)
»   return retval

messagetext = """Hello,

This is a *great* test message from Chapter 12.  I hope you enjoy it!

-- Anonymous"""
messagehtml = """Hello,<P>
This is a <B>great</B> test message from Chapter 12.  I hope you enjoy
it!<P>
-- <I>Anonymous</I>"""


msg = MIMEMultipart('alternative')
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 12'
msg['Date'] = utils.formatdate(localtime = 1)
msg['Message-ID'] = utils.make_msgid()

msg.attach(alternative(messagetext, 'text/plain'))
msg.attach(alternative(messagehtml, 'text/html'))
print msg.as_string()
```

Notice the differences between an alternative message and a message with attachments! With the alternative message, no `Content-Disposition` header is inserted. Also, the `MIMEMultipart` object is passed the `alternative` subtype to tell the mail reader that all objects in this multipart are alternative views of the same thing.

Note again that it is always most polite to include the plain-text object first for people with ancient or incapable mail readers, which simply show them the entire message as text! In fact, we ourselves will now view the message that way, by running it on the command line in Listing 12–11.

Listing 12–11. *What an Alternative-Part Message Looks Like*

```
$ ./mime_gen_alt.py
Content-Type: multipart/alternative; boundary="===============1543078954=="
MIME-Version: 1.0
To: recipient@example.com
From: Test Sender <sender@example.com>
Subject: Test Message, Chapter 12
Date: Thu, 11 Dec 2003 19:36:56 -0600
Message-ID: <20031212013656.21447.34593@user.example.com>
```

```
--===============1543078954==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hello,
This is a *great* test message from Chapter 12.  I hope you enjoy it!
-- Anonymous
--===============1543078954==
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Hello,<P>

This is a <B>great</B> test message from Chapter 12.  I hope you enjoy
it!<P>
-- <I>Anonymous</I>
--===============1543078954==--
```

An HTML-capable mail reader will choose the second view, and give the user a fancy representation of the message with the word "great" in bold and "Anonymous" in italics. A text-only reader will instead choose the first view, and the user will still at least see a readable message instead of one filled with angle brackets.

Composing Non-English Headers

Although you have seen how MIME can encode message body parts with base-64 to allow 8-bit data to pass through, that does not solve the problem of special characters in headers. For instance, if your name was Michael Müller (with an umlaut over the "u"), you would have trouble representing your name accurately in your own alphabet. The "u" would come out bare.

Therefore, MIME provides a way to encode data in headers. Take a look at Listing 12–12 for how to do it in Python.

Listing 12–12. *Using a Character Encoding for a Header*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - mime_headers.py
# This program requires Python 2.5 or above

from email.mime.text import MIMEText
from email.header import Header

message = """Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous"""

msg = MIMEText(message)
msg['To'] = 'recipient@example.com'
fromhdr = Header()
```

```
fromhdr.append(u"Michael M\xfcller")
fromhdr.append('<mmueller@example.com>')
msg['From'] = fromhdr
msg['Subject'] = 'Test Message, Chapter 12'

print msg.as_string()
```

The code `'\xfc'` in the Unicode string (strings in Python source files that are prefixed with `u` can contain arbitrary Unicode characters, rather than being restricted to characters whose value is between 0 and 255) represents the character 0xFC, which stands for “ü”. Notice that we build the address as two separate pieces, the first of which (the name) needs encoding, but the second of which (the e-mail address) can be included verbatim. Building the `From` header this way is important, so that the e-mail address winds up legible regardless of whether the user’s client can decode the fancy international text; take a look at Listing 12–13 for the result.

Listing 12–13. *Using a Character Encoding for a Header*

```
$ ./mime_headers.py
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: recipient@example.com
From: =?iso-8859-1?q?Michael_M=FCller?= <mmueller@example.com>
Subject: Test Message, Chapter 12
Date: Thu, 11 Dec 2003 19:37:56 -0600
Message-ID: <20031212013756.21447.34593@user.example.com>

Hello,

This is a test message from Chapter 12.  I hope you enjoy it!

-- Anonymous
```

Here is what would have happened if you had failed to build the `From` header from two different pieces, and instead tried to include the e-mail address along with the internationalized name:

```
>>> from email.header import Header
>>> h = u'Michael M\xfcller <mmueller@example.com>'
>>> print Header(h).encode()
=?utf-8?q?Michael_M=C3=BCller_=3Cmmueller=40example=2Ecom=3E?=
```

If you look very carefully, you can find the e-mail address in there somewhere, but certainly not in a form that a person —or their e-mail client —would find recognizable!

Composing Nested Multiparts

Now that you know how to generate a message with alternatives and one with attachments, you may be wondering how to do both. To do that, you create a standard multipart for the main message. Then you create a `multipart/alternative` inside that for your body text, and attach your message formats to it. Finally, you attach the various files. Take a look at Listing 12–14 for the complete solution.

Listing 12–14. *Doing MIME with Both Alternatives and Attachments*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - mime_gen_both.py

from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email import utils, encoders
import mimetypes, sys

def genpart(data, contenttype):
    maintype, subtype = contenttype.split('/')
    if maintype == 'text':
        retval = MIMEText(data, _subtype=subtype)
    else:
        retval = MIMEBase(maintype, subtype)
        retval.set_payload(data)
        encoders.encode_base64(retval)
    return retval


def attachment(filename):
    fd = open(filename, 'rb')
    mimetype, mimeencoding = mimetypes.guess_type(filename)
    if mimeencoding or (mimetype is None):
        mimetype = 'application/octet-stream'
    retval = genpart(fd.read(), mimetype)
    retval.add_header('Content-Disposition', 'attachment',
            filename = filename)
    fd.close()
    return retval

messagetext = """Hello,

This is a *great* test message from Chapter 12.  I hope you enjoy it!

-- Anonymous"""

messagehtml = """Hello,<P>
This is a <B>great</B> test message from Chapter 12.  I hope you enjoy
it!<P>
-- <I>Anonymous</I>"""

msg = MIMEMultipart()
msg['To'] = 'recipient@example.com'
msg['From'] = 'Test Sender <sender@example.com>'
msg['Subject'] = 'Test Message, Chapter 12'
msg['Date'] = utils.formatdate(localtime = 1)
msg['Message-ID'] = utils.make_msgid()

body = MIMEMultipart('alternative')
body.attach(genpart(messagetext, 'text/plain'))
```

```
body.attach(genpart(messagehtml, 'text/html'))
msg.attach(body)

for filename in sys.argv[1:]:
    msg.attach(attachment(filename))
print msg.as_string()
```

The output from this program is large, so I won't show it here. You should also know that there is no fixed limit to how deep message components may be nested, though there is rarely any reason to go deeper than is shown here.

Parsing MIME Messages

Python's email module can read a message from a file or a string, and generate the same kind of in-memory object tree that we were generating ourselves in the aforementioned listings. To understand the e-mail's content, all you have to do is step through its structure.

You can even make adjustments to the message (for instance, you can remove an attachment), and then generate a fresh version of the message based on the new tree. Listing 12–5 shows a program that will read in a message and display its structure by walking the tree.

Listing 12–15. *Walking a Complex Message*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - mime_structure.py
# This program requires Python 2.2.2 or above

import sys, email

def printmsg(msg, level = 0):
    prefix = "|  " * level
    prefix2 = prefix + "|"
    print prefix + "+ Message Headers:"
    for header, value in msg.items():
        print prefix2, header + ":", value
    if msg.is_multipart():
        for item in msg.get_payload():
            printmsg(item, level + 1)

msg = email.message_from_file(sys.stdin)
printmsg(msg)
```

This program is short and simple. For each object it encounters, it checks to see if it is multipart; if so, the children of that object are displayed as well. The output of this program will look something like this, given as input a message that contains a body in alternative form and a single attachment:

```
$ ./mime_gen_both.py /tmp/test.gz | ./mime_structure.py
+ Message Headers:
| Content-Type: multipart/mixed; boundary="===============1899932228=="
| MIME-Version: 1.0
| To: recipient@example.com
| From: Test Sender <sender@example.com>
| Subject: Test Message, Chapter 12
| Date: Fri, 12 Dec 2003 16:23:05 -0600
```

```
| Message-ID: <20031212222305.13361.15560@user.example.com>
|  + Message Headers:
|  | Content-Type: multipart/alternative; boundary="===============1287885775=="
|  | MIME-Version: 1.0
|  |  + Message Headers:
|  |  | Content-Type: text/plain; charset="us-ascii"
|  |  | MIME-Version: 1.0
|  |  | Content-Transfer-Encoding: 7bit
|  |  + Message Headers:
|  |  | Content-Type: text/html; charset="us-ascii"
|  |  | MIME-Version: 1.0
|  |  | Content-Transfer-Encoding: 7bit
|  + Message Headers:
|  | Content-Type: application/octet-stream
|  | MIME-Version: 1.0
|  | Content-Transfer-Encoding: base64
|  | Content-Disposition: attachment; filename="/tmp/test.gz"
```

Individual parts of a message can easily be extracted. You will recall that there are several ways that message data may be encoded; fortunately, the `email` module can decode them all! Listing 12–16 shows a program that will let you decode and save any component of a MIME message:

Listing 12–16. *Decoding Attachments in a MIME Message*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 12 - mime_decode.py
# This program requires Python 2.2.2 or above

import sys, email
counter = 0
parts = []

def printmsg(msg, level = 0):
    global counter
    l = "|  " * level
    if msg.is_multipart():
        print l + "Found multipart:"
        for item in msg.get_payload():
            printmsg(item, level + 1)
    else:
        disp = ['%d. Decodable part' % (counter + 1)]
        if 'content-type' in msg:
            disp.append(msg['content-type'])
        if 'content-disposition' in msg:
            disp.append(msg['content-disposition'])
        print l + ", ".join(disp)
        counter += 1
        parts.append(msg)

inputfd = open(sys.argv[1])
msg = email.message_from_file(inputfd)
printmsg(msg)

while 1:
```

```
»   print "Select part number to decode or q to quit: "
»   part = sys.stdin.readline().strip()
»   if part == 'q':
»   »   sys.exit(0)
»   try:
»   »   part = int(part)
»   »   msg = parts[part - 1]
»   except:
»   »   print "Invalid selection."
»   »   continue

»   print "Select file to write to:"
»   filename = sys.stdin.readline().strip()
»   try:
»   »   fd = open(filename, 'wb')
»   except:
»   »   print "Invalid filename."
»   »   continue

»   fd.write(msg.get_payload(decode = 1))
```

This program steps through the message, like the last example. We skip asking the user about message components that are multipart because those exist only to contain other message objects, like text and attachments; multipart sections have no actual payload of their own.

When run, the program looks something like this:

```
$ ./mime_decode.py testmessage.txt
Found multipart:
|  Found multipart:
|  |  1. Decodable part, text/plain; charset="us-ascii"
|  |  2. Decodable part, text/html; charset="us-ascii"
|  3. Decodable part, application/octet-stream, attachment; filename="/tmp/test.gz"
Select part number to decode or q to quit:
3
Select file to write to:
/tmp/newfile.gz
Select part number to decode or q to quit:
q
```

Decoding Headers

The last trick that we should cover regarding MIME messages is decoding headers that may have been encoded with foreign languages. The function `decode_header()` takes a single header and returns a list of pieces of the header; each piece is a binary string together with its encoding (named as a string if it is something besides 7-bit ASCII, else the value `None`):

```
>>> x = '=?iso-8859-1?q?Michael_M=FCller?= <mmueller@example.com>'
>>> import email.header
>>> pieces = email.header.decode_header(x)
>>> print pieces
[('Michael M\xfcller', 'iso-8859-1'), ('<mmueller@example.com>', None)]
```

Of course, this raw information is likely to be of little use to you. To instead see the actual text inside the encoding, use the `decode()` function of each binary string in the list (falling back to an 'ascii' encoding if `None` was returned) and paste the result together with spaces:

```
>>> print ' '.join( s.decode(enc or 'ascii') for s,enc in pieces )
Michael Müller <mmueller@example.com>
```

It is always good practice to use `decode_header()` on any of the "big three" headers —`From`, `To`, and `Subject` —before displaying them to the user. If no special encoding was used, then the result will simply be a one-element list containing the header string with a `None` encoding.

Summary

Traditional e-mail messages contain headers and a body. All parts of a traditional message must be represented using a 7-bit encoding, which generally prohibits the use of anything other than text using the Latin alphabet as used in English.

Headers provide useful information for mail reader programs and for people reading mail. Contrary to what many expect, except in special circumstances, the headers don't directly dictate where messages get sent.

Python's e-mail modules can both generate messages and parse messages. To generate a traditional message, an instance of `email.mime.text.MIMEText` or `email.message.Message` can be created. The `Date` and `Message-ID` headers are not added by default, but can be easily added using convenience functions.

To parse a traditional or MIME message, just call `email.message_from_file(fd)` where `fd` is the file descriptor from which to read its content. Parsing of `Date` headers can be tricky, but it is usually possible without too much difficulty.

MIME is a set of extensions to the e-mail format that permit things such as non-text data, attachments, alternative views of content, and different character sets. Multipart MIME messages can be used for attachments and alternative views, and are constructed in a "tree" fashion.

CHAPTER 13

SMTP

As we outlined at the beginning of the previous chapter, the actual movement of e-mail between systems is accomplished through SMTP: the "Simple Mail Transport Protocol." It was first defined in 1982 in RFC 821; the most recent RFC defining it is 5321. It typically serves in two roles:

- When a user types an e-mail message on a laptop or desktop machine, the e-mail client uses SMTP to *submit* the e-mail to a real server that can send it along to its destination.
- E-mail servers themselves use SMTP to *deliver* messages, sending them across the Internet to the server in charge of the recipient e-mail address's domain (the part of the e-mail address after the `@` sign).

There are several differences between how SMTP is used for submission and delivery. But before discussing them, we should quickly outline the difference between users who check e-mail with a local e-mail *client*, and people who instead use a *webmail* service.

E-mail Clients, Webmail Services

The role of SMTP in message *submission*, where the user presses "Send" and expects a message to go winging its way across the Internet, will probably be least confusing if we trace the history of how users have historically worked with Internet mail.

The key concept to understand as we begin this history is that users have *never* been asked to sit around and wait for an e-mail message to actually be delivered. This process can often take quite a bit of time—and up to several dozen repeated attempts—before an e-mail message is actually delivered to its destination. Any number of things could cause delays: a message could have to wait because other messages are already being transmitted across a link of limited bandwidth; the destination server might be down for a few hours, or its network might not be currently accessible because of a glitch; and if the mail is destined for a large organization, then it might have to make several different "hops" as it arrives at the big university server, then is directed to a smaller college e-mail machine, and then finally is directed to a departmental e-mail server.

So understanding what happens when the user hits "Send" is, essentially, to understand how the finished e-mail message gets submitted to the first of possibly several e-mail *queues* in which it can languish until the circumstances are just right for its delivery to occur (which we will discuss in the next section, on e-mail delivery).

In the Beginning Was the Command Line

The first generations of e-mail users were given usernames and passwords by their business or university that gave them command-line access to the large mainframes where user files and general-purpose programs were kept. These large machines typically ran an e-mail daemon that maintained an outgoing queue, right on the same box as the users who were busily typing messages into small command-line programs. Several such programs each had their heyday; `mail` was followed by the fancier `mailx`, which then fell to the far prettier interfaces—and great capabilities—of `elm`, `pine`, and finally `mutt`.

But for all of these early users, the network was not even involved in the simple task of e-mail submission; after all, the e-mail client and the server were on the same machine! The actual means of bridging this small gap and performing e-mail submission was a mere implementation detail, usually hidden behind a command-line client program that came with the server software and that knew exactly how to communicate with it. The first widespread e-mail daemon, `sendmail`, came with a program for submitting e-mail called `/usr/lib/sendmail`.

Because the first generation of client programs for reading and writing e-mail were designed to interact with `sendmail`, the mail daemons that have subsequently risen to popularity, like `qmail` and `postfix` and `exim`, generally followed suit by providing a `sendmail` binary of their own (its official home is now `/usr/sbin`, thanks to recent filesystem standards) that, when invoked by the user's e-mail program, would follow their own peculiar procedure for getting a message moved into the queue.

When e-mail arrived, it was typically deposited into a file belonging to the user to whom the message had been addressed. The e-mail client running on the command line could simply open this file and parse it to see the messages that were waiting for the user to read. This book does not cover these *mailbox formats*, because we have to keep our focus on how e-mail uses the network; but if you are curious, you can check out the `mailbox` package in the Python Standard Library, which supports all of the strange and curious ways in which various e-mail programs have read and written messages to disk over the years.

The Rise of Clients

The next generation of users to reach the Internet were often not familiar with the idea of a command line; they instead had experience with the graphical interface of an Apple Macintosh—or, when it later arrived, the Microsoft Windows operating system—and expected to accomplish things by clicking an icon and running a graphical program. So a number of different e-mail *clients* were written that brought this Internet service to the desktop; Mozilla Thunderbird and Microsoft Outlook are only two of the most popular of the clients still in use today.

The problems with this approach are obvious.

First, the problem of reading incoming e-mail was transformed from a simple task—your client program opened a file and read it—to being an operation that would require a network connection. When you brought your graphical desktop online, it somehow had to reach across the Internet to a full-time server that had been receiving e-mail on your behalf while you were away, and bring the mail to the local machine.

Second, users are notorious for not properly backing up their desktop and laptop file systems, and clients that downloaded and stored messages locally made those messages thereby vulnerable to obliteration when the laptop or desktop hard drive finally crashed; by contrast, university and industrial servers—despite their clunky command lines—usually had small armies of people specifically tasked with keeping their data archived, duplicated, and safe.

Third, laptop and desktop machines are usually not suitable environments for an e-mail server and its queue of outgoing messages. Users, after all, often turn their machines off when they are done using them; or they disconnect from the Internet; or they leave the Internet café and lose their wireless signal

anyway. Outgoing messages generally need more attention than this, so completed e-mails need some way to be *submitted* back to a full-time server for queuing and delivery.

But programmers are clever people, and they came up with a series of solutions to these problems.

First, new protocols were invented—first the Post Office Protocol, POP, which we discuss in Chapter 14, and then the Internet Message Access Protocol, IMAP, covered in Chapter 15—that let a user's e-mail client authenticate with a password and download mail from the full-time server that had been storing it. Passwords were necessary since, after all, you do not want the invention of a new protocol to suddenly make it easy for other people to connect to your ISP's servers and read your mail! This solved the first problem.

But what about the second problem, that of persistence: avoiding the loss of mail when desktop and laptop hard drives crash? This inspired two sets of advances. First, people using POP often learned to turn off its default mode, in which the e-mail on the server is deleted once is has been downloaded, and learned to leave copies of important mail on the server, from which they could fetch mail again later if they had to re-install their computer and start from scratch. Second, they started moving to IMAP, because—if their e-mail server chose to support this more advanced protocol—it meant that they could not only leave incoming e-mail messages on the server for safekeeping, but also arrange the messages in folders right there on the server! This let them use their e-mail client program as a mere window through which to see mail that remained stored on the server, rather than having to manage an e-mail storage area on their laptop or desktop itself.

Finally, how does e-mail make it back to the server when the user finishes writing an e-mail message and hits "Send"? This task—again, called e-mail "submission" in the official terminology—brings us back to the subject of this chapter: e-mail submission takes place using the SMTP protocol. But, as we shall see, there are usually two differences between SMTP as it is spoken between servers on the Internet and when it is used for client e-mail submission, and both differences are driven by the modern need to combat spam. First, because most ISPs block outgoing messages to port 25 from laptops and desktops so that these small machines cannot be hijacked by viruses and used as mail servers, e-mail submission is usually directed to port 587. Second, to prevent every spammer from connecting to your ISP and claiming that they want to send a message purportedly from you, e-mail clients use *authenticated SMTP* that includes the user's username and password.

Through these mechanisms, e-mail has been brought to the desktop. Both in large organizations like universities and businesses, and also in ISPs catering to users at home, it is still common to hand out instructions to each user that tell them to:

- Install an e-mail client like Thunderbird or Outlook
- Enter the hostname and protocol from which e-mail can be fetched
- Configure the outgoing server's name and SMTP port number
- Assign a username and password with which connections to both services can be authenticated

While e-mail clients can be cumbersome to configure and the servers can be difficult maintain, they were originally the only way that e-mail could be supported using a familiar graphical interface to the new breed of users staring at large colorful displays. And, today, they allow users an enviable freedom of choice: their ISP simply decides whether to support POP, or IMAP, or both, and the user (or, at least, the non-enterprise user!) is then free to try out the various e-mail clients and settle on the one that they like best.

The Move to Webmail

And, finally, yet another generational shift has occurred on the Internet.

Users once had to download and install a plethora of clients in order to experience all that the Internet had to offer; many older readers will remember having Windows or Mac machines on which they eventually installed client programs for such diverse protocols as Telnet, FTP, the Gopher directory service, Usenet newsgroups, and, when it came along, a World Wide Web browser. (Unix users typically found clients for each basic protocol already installed when they first logged in to a well-configured machine, though they might have chosen to install more advanced replacements for some of the programs, like `ncftp` in place of the clunky default FTP client.)

But, no longer!

The average Internet user today knows only a single client: their web browser. Thanks to the fact that web pages can now use JavaScript to respond and re-draw themselves as the user clicks and types, the Web is not only replacing all traditional Internet protocols—users browse and fetch files on web pages, not through FTP; they read message boards, rather than connecting to the Usenet—but it is also obviating the need for many traditional desktop clients. Why convince thousands of users to download and install a client, clicking through several warnings about how your software might harm their computer, if your application is one that could be offered through an interactive web page?

In fact, the web browser has become so preeminent that many Internet users are not even aware that they *have* a web browser. They therefore use the words "Internet" and "Web" interchangeably, and think that both terms refer to "all those documents and links that give me Facebook and YouTube and the Wikipedia." This obliviousness to the fact that they are viewing the Web's glory through some particular client program with a name and identity—say through the dingy pane of Internet Explorer—is a constant frustration to evangelists for alternatives like Firefox, Google Chrome, and Opera, who find it difficult to convince people to change from a program that they are not even aware they are using!

Obviously, if such users are to read e-mail, it must be presented to them on a web page, where they read incoming mail, sort it into folders, and compose and send replies. And so there exist many web sites offering e-mail services through the browser—Gmail and Yahoo! Mail being among the most popular—as well as server software, like the popular SquirrelMail, that system administrators can install if they want to offer webmail to users at their school or business.

What does this transition mean for e-mail protocols, and the network?

Interestingly enough, the webmail phenomenon essentially moves us *back* in time, to the simpler days when e-mail submission and e-mail reading were private affairs, confined to a single mainframe server and usually not using public protocols at all. Of course, these modern services—especially the ones run by large ISPs, and companies like Google and Yahoo!—must be gargantuan affairs, involving hundreds of servers at locations around the world; so, certainly, network protocols are doubtless involved at every level of e-mail storage and retrieval.

But the point is that these are now *private* transactions, internal to the organization running the webmail service. You browse e-mail in your web browser; you write e-mail using the same interface; and when you hit "Send," well, who knows what protocol Google or Yahoo! uses internally to pass the new message from the web server receiving your HTTP POST to a mail queue from which it can be delivered? It could be SMTP; it could be an in-house RPC protocol; or it could even be an operation on common filesystems to which the web and e-mail servers are connected.

For the purpose of this book, the important thing is that—unless you are an engineer working at such an organization—you will never see whether POP, or IMAP, or something else is at work, sitting behind the webmail interface and manipulating your messages.

E-mail browsing and submission, therefore, become a black box: your browser interacts with a web API, and on the other end, you will see plain old SMTP connections originating from and going to the large organization as mail is delivered in each direction. But in the world of webmail, client protocols are removed from the equation, taking us back to the old days of pure server-to-server unauthenticated SMTP.

How SMTP Is Used

The foregoing narrative has hopefully helped you structure your thinking about Internet e-mail protocols, and realize how they fit together in the bigger picture of getting messages to and from users.

But the subject of this chapter is a narrower one—the Simple Mail Transport Protocol in particular. And we should start by stating the basics, in the terms we learned in Part 1 of this book:

- SMTP is a TCP/IP-based protocol.
- Connections can be authenticated, or not.
- Connections can be encrypted, or not.

Most e-mail connections across the Internet these days seem to lack any attempt at encryption, which means that whoever owns the Internet backbone routers are theoretically in a position to read simply staggering amounts of other people's mail.

What are the two ways, given our discussion in the last section, that SMTP is used?

First, SMTP can be used for e-mail *submission* between a client e-mail program like Thunderbird or Outlook, claiming that a user wants to send e-mail, and a server at an organization that has given that user an e-mail address. These connections generally use authentication, so that spammers cannot connect and send millions of messages on a user's behalf without his or her password. Once received, the server puts the message in a queue for delivery (and often makes its first attempt at sending it moments later), and the client can forget about the message and presume the server will keep trying to deliver it.

Second, SMTP is used *between* Internet mail servers as they move e-mail from its origin to its destination. This typically involves no authentication; after all, big organizations like Google, Yahoo!, and Microsoft do not know the passwords of each other's users, so when Yahoo! receives an e-mail from Google claiming that it was sent from an `@gmail.com` user, Yahoo! just has to believe them (or not—sometimes organizations blacklist each other if too much spam is making it through their servers, as happened to a friend of mine the other day when Hotmail stopped accepting his client's newsletters from GoDaddy's servers because of alleged problems with spam).

So, typically, no authentication takes place between servers talking SMTP to each other—and even encryption against snooping routers seems to be used only rarely.

Because of the problem of spammers connecting to e-mail servers and claiming to be delivering mail from another organization's users, there has been an attempt made to lock down who can send e-mail on an organization's behalf. Though controversial, some e-mail servers consult the Sender Policy Framework (SPF), defined in RFC 4408, to see whether the server they are talking to really has the authority to deliver the e-mails it is transmitting.

But the SPF and other anti-spam technologies are unfortunately beyond the scope of this book, which must limit itself to the question of using the basic protocols themselves from Python. So we now turn to the more technical question of how you will actually use SMTP from your Python programs.

Sending E-Mail

Before proceeding to share with you the gritty details of the SMTP protocol, one warning is in order: if you are writing an interactive program, daemon, or web site that needs to send e-mail, then your site or system administrator (in cases where that is not you!) might have an opinion about how your program sends mail—and they might save you a *lot* of work by doing so!

As noted in the introduction, successfully sending e-mail generally requires a queue where a message can sit for seconds, minutes, or days until it can be successfully transmitted toward its destination. So you typically do *not* want your programs using Python's `smtplib` to send mail directly to a message's destination—because if your first transmission attempt fails, then you will be stuck with the

job of writing a full “mail transfer agent” (MTA), as the RFCs call an e-mail server, and give it a full standards-compliant re-try queue. This is not only a big job, but also one that has already been done well several times, and you will be wise to take advantage of one of the existing MTAs (look at `postfix`, `exim`, and `qmail`) before trying to write something of your own.

So only rarely will you be making SMTP connections out into the world from Python. More usually, your system administrator will tell you one of two things:

- That you should make an authenticated SMTP connection to an existing e-mail server, using a username and password that will belong to your application, and give it permission to use the e-mail server to queue outgoing messages
- That you should run a local binary on the system—like the `sendmail` program—that the system administrator has already gone to the trouble to configure so that local programs can send mail

As of late 2010, the Python Library FAQ has sample code for invoking a `sendmail` compatible program; take a look at the section “How do I send mail from a Python script?” on the following page:

`http://docs.python.org/faq/library.html`

Since this book is about networking, we will not cover this possibility in detail, but you should remember to do raw SMTP yourself only when no simpler mechanism exists on your machine for sending e-mail.

Headers and the Envelope Recipient

The key concept involved in SMTP that consistently confuses beginners is that the addressee headers you are so familiar with—`To`, `Cc` (carbon copy), and `Bcc` (blind carbon copy)—are *not* consulted by the SMTP protocol to decide where your e-mail goes!

This surprises many users. After all, almost every e-mail program in existence asks you to fill in those addressee fields, and when you hit “Send,” the message wings it way out to those mailboxes. What could be more natural? But it turns out that this is a feature of the e-mail client itself, not of the SMTP protocol: the protocol knows only that each message has an “envelope” around it naming a sender and some recipients. SMTP itself does not care whether those names are ones that it can find in the headers of the message.

That e-mail must work this way will actually be quite obvious if you think for a moment about the `Bcc` blind carbon-copy header. Unlike the `To` and `Cc` headers, which make it to the e-mail’s destination and let each recipient see who else was sent that e-mail, the `Bcc` header names people who you want to receive the mail *without* any of the other recipients knowing. Blind copies let you quietly bring a message to someone’s attention without alerting the other readers of the e-mail.

The existence of a header like `Bcc` that can be present when you compose a message but disappear as it is sent raises two points:

- Your e-mail client edits your message’s headers before sending it. Besides removing the `Bcc` header so that none of the e-mail's recipients gets a copy of it, the client typically adds headers as well, such as a unique message ID, and perhaps the name of the e-mail client itself (an e-mail open on my desktop right now, for example, identifies the `X-Mailer` that sent it as “YahooMailClassic”).
- An e-mail can pass across SMTP toward a destination address that is mentioned *nowhere* in the e-mail headers or text itself—and can do this for the most legitimate of reasons.

This mechanism also helps support mailing lists, so that an e-mail whose `To` says `advocacy@python.org` can actually be delivered, without rewritten headers, to the dozens or hundreds of people who subscribe to that list.

So, as you read the following descriptions of SMTP, keep reminding yourself that the headers-plus-body that make up the e-mail message itself are separate from the "envelope sender" and "envelope recipient" that will be mentioned in the protocol descriptions. Yes, it is true that your e-mail client, whether you are using `/usr/sbin/sendmail` or Thunderbird or Google Mail, probably asked you for the recipient's e-mail address only once; but it then proceeded to use it in *two* different places, once in the `To` header at the top of the message, and then again "outside" of the message when it spoke SMTP in order to send the e-mail on its way.

Multiple Hops

Once upon a time, e-mail often traveled over only one SMTP "hop" between the mainframe on which it was composed to the machine on whose disk the recipient's in-box was stored. These days, messages often travel through a half-dozen servers or more before reaching their destination. This means that the SMTP envelope recipient, described in the last section, repeatedly changes as the message nears its destination.

An example should make this clear. Several of the following details are fictitious, but they should give you a good idea of how messages actually traverse the Internet.

Imagine a worker in the central IT organization at Georgia Tech who tells his friend that his e-mail address is `brandon@gatech.edu`. When the friend later sends him a message, the friend's e-mail provider will look up the domain `gatech.edu` in the Domain Name Service (DNS; see Chapter 4), receive a series of `MX` records in reply, and connect to one of those IP address to deliver the message. Simple enough, right?

But the server for `gatech.edu` serves an entire campus! To find out where `brandon` is, it consults a table, finds his department, and learns that his official e-mail address is actually:

```
brandon.rhodes@oit.gatech.edu
```

So the `gatech.edu` server in turn does a DNS lookup of `oit.gatech.edu` and then uses SMTP—the message's second SMTP hop, if you are counting—to send the message to the e-mail server for OIT, the Office of Information Technology.

But OIT long ago abandoned their single-server solution that used to keep all of their mail on a single Unix server. Instead, they now run a sophisticated e-mail solution that users can access through webmail, POP, and IMAP. Incoming mail arriving at `oit.gatech.edu` is first sent randomly to one of several spam-filtering servers (third hop), say, the server named `spam3.oit.gatech.edu`. Then it is handed off randomly to one of eight redundant e-mail servers, and so after the fourth hop, the message is in the queue on `mail7.oit.gatech.edu`.

We are almost done: the routing servers like `mail7` are the ones with access to the lookup tables of which back-end mailstores, connected to large RAID arrays, hold which users. So `mail7` does an LDAP lookup for `brandon.rhodes`, concludes that his mail lives on the `anvil.oit.gatech.edu` server, and in a fifth and final SMTP hop, the mail is delivered to `anvil` and there is written to the redundant disk array.

That is why e-mail often takes at least a few seconds to traverse the Internet: large organizations and big ISPs tend to have several levels of servers that a message must negotiate before its delivery.

How can you find out what an e-mail's route was? It was emphasized previously that the SMTP protocol does *not* look inside e-mail headers, but has its own idea about where a message should be going—that, as we have just seen, can change with every hop that a message makes toward its destination. But it turns out that e-mail servers *are* encouraged to *write* new headers, precisely to keep track of a message's circuitous route from its original to its destination.

These headers are called `Received` headers, and they are a gold mine for confused system administrators trying to debug problems with their mail systems. Take a look at any e-mail message, and ask your mail client to display all of the headers; you should be able to see every step that the message

took toward its destination. (An exception is spam messages: spammers often write several fictitious Received headers at the top of their messages to make it look like the message has originated from a reputable organization.) Finally, there is probably a Delivered-to header that is written when the last server in the chain is finally able to triumphantly write the message to physical storage in someone's mailbox.

Because each server tends to add its Received header to the *top* of the e-mail message—this saves time, and prevents each server from having to search to the bottom of the Received headers that have been written so far—you should read them "backward": the oldest Received header will be the one listed last, and as you read up the screen toward the top, you will be following the e-mail from its origin to its destination. Try it: bring up a recent e-mail message you have received, select its "View All Message Headers" or "Show Original" option, and look for the received headers near the top. Did the message require more, or fewer, steps to reach your in-box than you would have expected?

Introducing the SMTP Library

Python's built-in SMTP implementation is in the Python Standard Library module smtplib, which makes it easy to do simple tasks with SMTP.

In the examples that follow, the programs are designed to take several command-line arguments: the name of an SMTP server, a sender address, and one or more recipient addresses. Please use them cautiously; name only an SMTP server that you yourself run or that you know will be happy receiving your test messages, lest you wind up getting an IP address banned for sending spam!

If you don't know where to find an SMTP server, you might try running a mail daemon like postfix or exim locally and then pointing these example programs at localhost. Many UNIX, Linux, and Mac OS X systems have an SMTP server like one of these already listening for connections from the local machine.

Otherwise, consult your network administrator or Internet provider to obtain a proper hostname and port. Note that you usually cannot just pick a mail server at random; many store or forward mail only from certain authorized clients.

So, take a look at Listing 13–1 for a very simple SMTP program!

Listing 13–1. *Sending E-mail with smtplib.sendmail()*

```
#!/usr/bin/env python
# Basic SMTP transmission - Chapter 13 - simple.py

import sys, smtplib

if len(sys.argv) < 4:
»   print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
»   sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the simple.py program
in Foundations of Python Network Programming.
""" % (', '.join(toaddrs), fromaddr)
```

```
s = smtplib.SMTP(server)
s.sendmail(fromaddr, toaddrs, message)

print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

This program is quite simple, because it uses a very powerful and general function from inside the Standard Library.

It starts by generating a simple message from the user's command-line arguments (for details on generating fancier messages that contain elements beyond simple plain text, see Chapter 12). Then it creates an `smtplib.SMTP` object that connects to the specified server. Finally, all that's required is a call to `sendmail()`. If that returns successfully, then you know that the message was sent.

As was promised in the earlier sections of this chapter, you can see that the idea of who receives the message—the "envelope recipient"—is, down at this level, separate from the actual text of the message. This particular program writes a `To` header that happens to contain the same addresses to which it is sending the message; but the `To` header is just a piece of text, and could instead say anything else instead. (Whether that "anything else" would be willingly displayed by the recipient's e-mail client, or cause a server along the way to discard the message as spam, is another question!)

When you run the program, it will look like this:

```
$ ./simple.py localhost sender@example.com recipient@example.com
Message successfully sent to 2 recipient(s)
```

Thanks to the hard work that the authors of the Python Standard Library have put into the `sendmail()` method, it might be the only SMTP call you ever need! But to understand the steps that it is taking under the hood to get your message delivered, let's delve in more detail into how SMTP works.

Error Handling and Conversation Debugging

There are several different exceptions that might be raised while you're programming with `smtplib`. They are:

- `socket.gaierror` for errors looking up address information
- `socket.error` for general I/O and communication problems
- `socket.herror` for other addressing errors
- `smtplib.SMTPException` or a subclass of it for SMTP conversation problems

The first three errors are covered in more detail in Chapter 3; they are passed straight through the `smtplib` module and up to your program. But so long as the underlying TCP socket works, all problems that actually involve the SMTP e-mail conversation will result in an `smtplib.SMTPException`.

The `smtplib` module also provides a way to get a series of detailed messages about the steps it takes to send an e-mail. To enable that level of detail, you can call

```
smtpobj.set_debuglevel(1)
```

With this option, you should be able to track down any problems. Take a look at Listing 13–2 for an example program that provides basic error handling and debugging.

Listing 13–2. *A More Cautious SMTP Client*

```
#!/usr/bin/env python
# SMTP transmission with debugging - Chapter 13 - debug.py

import sys, smtplib, socket

if len(sys.argv) < 4:
»   print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
»   sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the debug.py program
in Foundations of Python Network Programming.
""" % (', '.join(toaddrs), fromaddr)

try:
»   s = smtplib.SMTP(server)
»   s.set_debuglevel(1)
»   s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror,
»   »   smtplib.SMTPException), e:
»   print " *** Your message may not have been sent!"
»   print e
»   sys.exit(1)
else:
»   print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

This program looks similar to the last one. However, the output will be very different; take a look at Listing 13–3 for an example.

Listing 13–3. *Debugging Output from `smtplib`*

```
$ ./debug.py localhost foo@example.com jgoerzen@complete.org
send: 'ehlo localhost\r\n'
reply: '250-localhost\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-SIZE 20480000\r\n'
reply: '250-VRFY\r\n'
reply: '250-ETRN\r\n'
reply: '250-STARTTLS\r\n'
reply: '250-XVERP\r\n'
reply: '250 8BITMIME\r\n'
reply: retcode (250); Msg: localhost
PIPELINING
SIZE 20480000
VRFY
```

```
ETRN
STARTTLS
XVERP
8BITMIME
send: 'mail FROM:<foo@example.com> size=157\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
send: 'rcpt TO:<jgoerzen@complete.org>\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
send: 'data\r\n'
reply: '354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: End data with <CR><LF>.<CR><LF>
data: (354, 'End data with <CR><LF>.<CR><LF>')
send: 'To: jgoerzen@complete.org\r\n
From: foo@example.com\r\n
Subject: Test Message from simple.py\r\n
\r\n
Hello,\r\n
\r\n
This is a test message sent to you from simple.py and smtplib.
\r\n.
\r\n'
reply: '250 Ok: queued as 8094C18C0\r\n'
reply: retcode (250); Msg: Ok: queued as 8094C18C0
data: (250, 'Ok: queued as 8094C18C0')
Message successfully sent to 1 recipient(s)
```

From this example, you can see the conversation that `smtplib` is having with the SMTP server over the network. As you implement code that uses more advanced SMTP features, the details shown here will be more important, so let's look at what's happening.

First, the client (the `smtplib` library) sends an `EHLO` command (an "extended" successor to a more ancient command that was named, more readably, `HELO`) with your hostname in it. The remote server responds with its hostname, and lists any optional SMTP features that it supports.

Next, the client sends the `mail from` command, which states the "envelope sender" e-mail address and the size of the message. The server at this moment has the opportunity to reject the message (for example, because it thinks you are a spammer); but in this case, it responds with `250 Ok`. (Note that in this case, the code `250` is what matters; the remaining text is just a human-readable comment and varies from server to server.)

Then the client sends a `rcpt to` command, with the "envelope recipient" that we talked so much about earlier in this chapter; you can finally see that, indeed, it is transmitted separately from the text of the message itself when using the SMTP protocol. If you were sending the message to more than one recipient, they would each be listed on the `rcpt to` line.

Finally, the client sends a `data` command, transmits the actual message (using verbose carriage-return-linefeed line endings, you will note, per the Internet e-mail standard), and finishes the conversation.

The `smtplib` module is doing all this automatically for you in this example. In the rest of the chapter, we will look at how to take more control of the process so you can take advantage of some more advanced features.

■ **Caution** Do not get a false sense of confidence because no error was detected during this first hop, and think that the message is now guaranteed to be delivered. In many cases, a mail server may accept a message, only to have delivery fail at a later time; read back over the foregoing “Multiple Hops” section, and imagine how many possibilities of failure there are before that message reaches its destination!

Getting Information from EHLO

Sometimes it is nice to know about what kind of messages a remote SMTP server will accept. For instance, most SMTP servers have a limit on what size message they permit, and if you fail to check first, then you may transmit a very large message only to have it rejected when you have completed transmission.

In the original version of SMTP, a client would send a `HELO` command as the initial greeting to the server. A set of extensions to SMTP, called ESMTP, has been developed to allow more powerful conversations. ESMTP-aware clients will begin the conversation with `EHLO`, which signals an ESMTP-aware server to send extended information. This extended information includes the maximum message size, along with any optional SMTP features that it supports.

However, you must be careful to check the return code. Some servers do not support ESMTP. On those servers, `EHLO` will just return an error. In that case, you must send a `HELO` command instead.

In the previous examples, we used `sendmail()` immediately after creating our SMTP object, so `smtplib` had to send its own “hello” message to the server. But if it sees you attempt to send the `EHLO` or `HELO` command on your own, then `sendmail()` will no longer attempt to send these commands itself.

Listing 13–4 shows a program that gets the maximum size from the server, and returns an error before sending if a message would be too large.

Listing 13–4. *Checking Message Size Restrictions*

```
#!/usr/bin/env python
# SMTP transmission with manual EHLO - Chapter 13 - ehlo.py

import sys, smtplib, socket

if len(sys.argv) < 4:
»   print "usage: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
»   sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the ehlo.py program
in Foundations of Python Network Programming.
""" % (', '.join(toaddrs), fromaddr)

try:
```

```
»   s = smtplib.SMTP(server)
»   code = s.ehlo()[0]
»   uses_esmtp = (200 <= code <= 299)
»   if not uses_esmtp:
»   »   code = s.helo()[0]
»   »   if not (200 <= code <= 299):
»   »   »   print "Remote server refused HELO; code:", code
»   »   »   sys.exit(1)

»   if uses_esmtp and s.has_extn('size'):
»   »   print "Maximum message size is", s.esmtp_features['size']
»   »   if len(message) > int(s.esmtp_features['size']):
»   »   »   print "Message too large; aborting."
»   »   »   sys.exit(1)

»   s.sendmail(fromaddr, toaddrs, message)

except (socket.gaierror, socket.error, socket.herror,
»   »   smtplib.SMTPException), e:
»   print " *** Your message may not have been sent!"
»   print e
»   sys.exit(1)
else:
»   print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

If you run this program, and the remote server provides its maximum message size, then the program will display the size on your screen and verify that its message does not exceed that size before sending. (For a tiny message like this, the check is obviously silly, but the listing shows you the pattern that you can use successfully with much larger messages.)

Here is what running this program might look like:

```
$ ./ehlo.py localhost foo@example.com jgoerzen@complete.org Maximum message size is 10240000
Message successfully sent to 1 recipient(s)
```

Take a look at the part of the code that verifies the result from a call to `ehlo()` or `helo()`. Those two functions return a list; the first item in the list is a numeric result code from the remote SMTP server. Results between 200 and 299, inclusive, indicate success; everything else indicates a failure. Therefore, if the result is within that range, you know that the server processed the message properly.

■ **Caution** The same caution as before applies here. The fact that the first SMTP server accepts the message does not mean that it will actually be delivered; a later server may have a more restrictive maximum size.

Besides message size, other ESMTP information is available as well. For instance, some servers may accept data in raw 8-bit mode if they provide the `8BITMIME` capability. Others may support encryption, as described in the next section. For more on ESMTP and its capabilities, which may vary from server to server, consult RFC 1869 or your own server's documentation.

Using Secure Sockets Layer and Transport Layer Security

As we discussed, e-mails sent in plain text over SMTP can be read by anyone with access to an Internet gateway or router across which the packets happen to pass. The best solution to this problem is to encrypt each e-mail with a public key whose private key is possessed only by the person to whom you are sending the e-mail; there are freely available systems such as PGP and GPG for doing exactly this. But regardless of whether the messages themselves are protected, individual SMTP conversations between particular pairs of machines can be encrypted and authenticated using a method known as SSL/TLS. In this section, you will learn about how SSL/TLS fits in with SMTP conversations.

Keep in mind that TLS protects only the SMTP "hops" that choose to use it—if you carefully use TLS to send an e-mail to a server, you have no control over whether that server uses TLS again if it has to forward your e-mail across another hop toward its destination.

For more details on TLS, please see Chapter 6; the code presented in this chapter cannot protect you from delivering a message to a fraudulent server without the certificate-handling described there.

The general procedure for using TLS in SMTP is as follows:

1. Create the SMTP object, as usual.
2. Send the `EHLO` command. If the remote server does not support `EHLO`, then it will not support TLS.
3. Check `s.has_extn()` to see if `starttls` is present. If not, then the remote server does not support TLS and the message can only be sent normally, in the clear.
4. Call `starttls()` to initiate the encrypted channel.
5. Call `ehlo()` a second time; this time, it's encrypted.
6. Finally, send your message.

The first question you have to ask yourself when working with TLS is whether you should return an error if TLS is not available. Depending on your application, you might want to raise an error for any of the following:

- There is no support for TLS on the remote side.
- The remote side fails to establish a TLS session properly.
- The remote server presents a certificate that cannot be validated.

Let us step through each of these scenarios and see when they may deserve an error message.

First, it is sometimes appropriate to treat a lack of support for TLS altogether as an error. This could be the case if you are writing an application that speaks to only a limited set of mail servers—perhaps mail servers run by your company that you know should support TLS, or mail servers run by a bank that you know supports TLS.

But since only a minority of mail servers on the Internet today support TLS, a mail program should not, in general, treat its absence as an error. Many TLS-aware SMTP clients will use TLS if available, but will fall back on standard, unsecured transmission otherwise. This is known as *opportunistic encryption* and is less secure than forcing all communications to be encrypted, but protects messages when the capability is present.

Second, sometimes a remote server claims to be TLS-aware but then fails to properly establish a TLS connection. This is often due to a misconfiguration on the server's end. To be as robust as possible, you may wish to retry your transmission to such a server with a new connection that you do not even try to encrypt.

Third, there is the situation where you cannot completely authenticate the remote server. Again, for a complete discussion of peer validation, see Chapter 6. If your security policy dictates that you must

exchange mail only with trusted servers, then lack of authentication is clearly a problem warranting an error message; but for a general-purpose client, it probably merits a warning instead.

Listing 13–5 acts as a TLS-capable general-purpose client. It will connect to a server and use TLS if it can; otherwise, it will fall back and send the message as usual. (But it *will* die with an error if the attempt to start TLS fails while talking to an ostensibly capable server!)

Listing 13–5. *Using TLS Opportunistically*

```
#!/usr/bin/env python
# SMTP transmission with TLS - Chapter 13 - tls.py

import sys, smtplib, socket

if len(sys.argv) < 4:
    print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
    sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,

This is a test message sent to you from the tls.py program
in Foundations of Python Network Programming.
""" % (', '.join(toaddrs), fromaddr)
try:
    s = smtplib.SMTP(server)
    code = s.ehlo()[0]
    uses_esmtp = (200 <= code <= 299)
    if not uses_esmtp:
        code = s.helo()[0]
        if not (200 <= code <= 299):
            print "Remove server refused HELO; code:", code
            sys.exit(1)
    if uses_esmtp and s.has_extn('starttls'):
        print "Negotiating TLS...."
        s.starttls()
        code = s.ehlo()[0]
        if not (200 <= code <= 299):
            print "Couldn't EHLO after STARTTLS"
            sys.exit(5)
        print "Using TLS connection."
    else:
        print "Server does not support TLS; using normal connection."
    s.sendmail(fromaddr, toaddrs, message)

except (socket.gaierror, socket.error, socket.herror,
        smtplib.SMTPException), e:
    print " *** Your message may not have been sent!"
    print e
    sys.exit(1)
```

```
else:
»   print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

If you run this program and give it a server that understands TLS, the output will look like this:

```
$ ./tls.py localhost jgoerzen@complete.org jgoerzen@complete.org
Negotiating TLS....
Using TLS connection.
Message successfully sent to 1 recipient(s)
```

Notice that the call to `sendmail()` in these last few listings is the same, regardless of whether TLS is used. Once TLS is started, the system hides that layer of complexity from you, so you do not need to worry about it. Please note that this TLS example is not fully secure, because it does not perform certificate validation; again, see Chapter 6 for details.

Authenticated SMTP

Finally, we reach the topic of Authenticated SMTP, where your ISP, university, or company e-mail server needs you to log in with a username and password to prove that you are not a spammer before they allow you to send e-mail.

For maximum security, TLS should be used in conjunction with authentication; otherwise your password (and username, for that matter) will be visible to anyone observing the connection. The proper way to do this is to establish the TLS connection first, and then send your authentication information only over the encrypted communications channel.

But using authentication itself is simple; `smtplib` provides a `login()` function that takes a username and a password. Listing 13–6 shows an example. To avoid repeating code already shown in previous listings, this listing does *not* take the advice of the previous paragraph, and sends the username and password over an un-authenticated connection that will send them in the clear.

Listing 13–6. *Authenticating over SMTP*

```
#!/usr/bin/env python
# SMTP transmission with authentication - Chapter 13 - login.py

import sys, smtplib, socket
from getpass import getpass

if len(sys.argv) < 4:
»   print "Syntax: %s server fromaddr toaddr [toaddr...]" % sys.argv[0]
»   sys.exit(2)

server, fromaddr, toaddrs = sys.argv[1], sys.argv[2], sys.argv[3:]

message = """To: %s
From: %s
Subject: Test Message from simple.py

Hello,
This is a test message sent to you from the login.py program
in Foundations of Python Network Programming.
""" % (', '.join(toaddrs), fromaddr)

sys.stdout.write("Enter username: ")
```

```
username = sys.stdin.readline().strip()
password = getpass("Enter password: ")

try:
»   s = smtplib.SMTP(server)
»   try:
»   »   s.login(username, password)
»   except smtplib.SMTPException, e:
»   »   print "Authentication failed:", e
»   »   sys.exit(1)
»   s.sendmail(fromaddr, toaddrs, message)
except (socket.gaierror, socket.error, socket.herror,
»   »   smtplib.SMTPException), e:
»   print " *** Your message may not have been sent!"
»   print e
»   sys.exit(1)
else:
»   print "Message successfully sent to %d recipient(s)" % len(toaddrs)
```

Most outgoing e-mail servers on the Internet do not support authentication. If you are using a server that does not support authentication, you will receive an "Authentication failed" error message from the `login()` attempt. You can prevent that by checking `s.has_extn('auth')` after calling `s.ehlo()` if the remote server supports ESMTP.

You can run this program just like the previous examples. If you run it with a server that does support authentication, you will be prompted for a username and password. If they are accepted, then the program will proceed to transmit your message.

SMTP Tips

Here are some tips to help you implement SMTP clients:

- There is no way to guarantee that a message was delivered. You can sometimes know immediately that your attempt failed, but the lack of an error does not mean that something else will not go wrong before the message is safely delivered to the recipient.
- The `sendmail()` function raises an exception if *any* of the recipients failed, though the message may still have been sent to other recipients. Check the exception you get back for more details. If it is very important for you to know specifics of which addresses failed—say, because you will want to try re-transmitting later without producing duplicate copies for the people who have already received the message—you may need to call `sendmail()` individually for each recipient. This is not generally recommended, however, since it will cause the message body to be transmitted multiple times.
- SSL/TLS is insecure without certificate validation; until validation happens, you could be talking to any old server that has temporarily gotten control of the normal server's IP address. To support certificate verification, the `starttls()` function takes some of the same arguments as `socket.ssl()`, which is described in Chapter 6. See the Standard Library documentation of `starttls()` for details.

- Python's `smtplib` is not meant to be a general-purpose mail relay. Rather, you should use it to send messages to an SMTP server close to you that will handle the actual delivery of mail.

Summary

SMTP is used to transmit e-mail messages to mail servers. Python provides the `smtplib` module for SMTP clients to use. By calling the `sendmail()` method of SMTP objects, you can transmit messages. The sole way of specifying the actual recipients of a message is with parameters to `sendmail()`; the `To`, `Cc`, and `Bcc` message headers are separate from the actual list of recipients.

Several different exceptions could be raised during an SMTP conversation. Interactive programs should check for and handle them appropriately.

ESMTP is an extension to SMTP. It lets you discover the maximum message size supported by a remote SMTP server prior to transmitting a message.

ESMTP also permits TLS, which is a way to encrypt your conversation with a remote server. Fundamentals of TLS are covered in Chapter 6.

Some SMTP servers require authentication. You can authenticate with the `login()` method.

SMTP does *not* provide functions for downloading messages from a mailbox to your own computer. To accomplish that, you will need the protocols discussed in the next two chapters. POP, discussed in Chapter 14, is a simple way to download messages. IMAP, discussed in Chapter 15, is a more capable and powerful protocol.

CHAPTER 14

POP

POP, the Post Office Protocol, is a simple protocol that is used to download e-mail from a mail server, and is typically used through an e-mail client like Thunderbird or Outlook. You can read the first few sections of Chapter 13 if you want the big picture of where e-mail clients, and protocols like POP, fit into the history of Internet mail.

The most common implementation of POP is version 3, and is commonly referred to as POP3. Because version 3 is so dominant, the terms POP and POP3 are practically interchangeable today.

POP's chief benefit—and also its biggest weakness—is its simplicity. If you simply need to access a remote mailbox, download any new mail that has appeared, and maybe delete the mail after the download, then POP will be perfect for you. You will be able to accomplish this task quickly, and without complex code.

However, this whole scheme has important limitations. POP does not support multiple mailboxes on the remote side, nor does it provide any reliable, persistent message identification. This means that you cannot use POP as a protocol for mail synchronization, where you leave the original of each e-mail message on the server while making a copy to read locally, because when you return to the server later you cannot easily tell which messages you have already downloaded. If you need this feature, you should check out IMAP, which is covered in Chapter 15.

The Python Standard Library provides the `poplib` module, which provides a convenient interface for using POP. In this chapter, you will learn how to use `poplib` to connect to a POP server, gather summary information about a mailbox, download messages, and delete the originals from the server. Once you know how to complete these four tasks, you will have covered all of the standard POP features!

Compatibility Between POP Servers

POP servers are often notoriously bad at correctly following standards. Standards also simply do not exist for some POP behaviors, so these details are left up to the authors of server software. So basic operations will generally work fine, but certain behaviors can vary from server to server.

For instance, some servers will mark all of your messages as read *whenever* you connect to the server—whether you download any of them or not!—while other servers will mark a given message as read only when it is downloaded. Some servers, on the other hand, never mark any messages as read at all. The standard itself seems to assume the latter behavior, but is not clear either way. Keep these differences in mind as you read this chapter.

Connecting and Authenticating

POP supports several authentication methods. The two most common are basic username-password authentication, and APOP, which is an optional extension to POP that helps protect passwords from being sent in plain-text if you are using an ancient POP server that does not support SSL.

The process of connecting and authenticating to a remote server looks like this in Python:

1. Create a POP3_SSL or just a plain POP3 object, and pass the remote hostname and port to it.
2. Call user() and pass_() to send the username and password. Note the underscore in pass_()! It is present because pass is a keyword in Python and cannot be used for a method name.
3. If the exception poplib.error_proto is raised, it means that the login has failed and the string value of the exception contains the error explanation sent by the server.

The choice between POP3 and POP3_SSL is governed by whether your e-mail provider offers—or, in this day and age, even requires—that you connect over an encrypted connection. Consult Chapter 6 for more information about SSL, but the general guideline should be to use it whenever it is at all feasible for you to do so.

Listing 14–1 uses the foregoing steps to log in to a remote POP server. Once connected, it calls stat(), which returns a simple tuple giving the number of messages in the mailbox and the messages' total size. Finally, the program calls quit(), which closes the POP connection.

Listing 14–1. *A Very Simple POP Session*

```
#!/usr/bin/env python
# POP connection and authentication - Chapter 14 - popconn.py

import getpass, poplib, sys

if len(sys.argv) != 3:
»   print 'usage: %s hostname user' % sys.argv[0]
»   exit(2)

hostname, user = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3_SSL(hostname)  # or "POP3" if SSL is not supported
try:
»   p.user(user)
»   p.pass_(passwd)
except poplib.error_proto, e:
»   print "Login failed:", e
else:
»   status = p.stat()
»   print "You have %d messages totaling %d bytes" % status
finally:
»   p.quit()
```

You can test this program if you have a POP account somewhere. Most people do—even large webmail services like GMail provide POP as an alternate means of checking your mailbox.

Run the preceding program, giving it two command-line arguments: the hostname of your POP server, and your username. If you do not know this information, contact your Internet provider or network administrator; note that on some services your username will be a plain string (like guido), whereas on others it will be your full e-mail address (guido@example.com).

The program will then prompt you for your password. Finally, it will display the mailbox status, without touching or altering any of your mail.

■ **Caution!** While this program does not alter any messages, some POP servers will nonetheless alter mailbox flags simply because you connected. Running the examples in this chapter against a live mailbox could cause you to lose information about which messages are read, unread, new, or old. Unfortunately, that behavior is server-dependent, and beyond the control of POP clients. I *strongly* recommend running these examples against a test mailbox rather than your live mailbox!

Here is how you might run the program:

```
$ ./popconn.py pop.example.com guido
Password: (type your password)
You have 3 messages totaling 5675 bytes
```

If you see output like this, then your first POP conversation has taken place successfully!

When POP servers do not support SSL to protect your connection from snooping, they sometimes at least support an alternate authentication protocol called APOP, which uses a challenge-response scheme to assure that your password is not sent in the clear. (But all of your e-mail will still be visible to any third party watching the packets go by!) The Python Standard Library makes this very easy to attempt: just call the `apop()` method, then fall back to basic authentication if the POP server you are talking to does not understand.

To use APOP but fall back to plain authentication, you could use a stanza like the one shown in Listing 14–2 inside your POP program (like Listing 14–1).

Listing 14–2. *Attempting APOP and Falling Back*

```
print "Attempting APOP authentication..."
try:
»   p.apop(user, passwd)
except poplib.error_proto:
»   print "Attempting standard authentication..."
»   try:
»   »   p.user(user)
»   »   p.pass_(passwd)
»   except poplib.error_proto, e:
»   »   print "Login failed:", e
»   »   sys.exit(1)
```

■ **Caution!** As soon as a login succeeds by whatever method, some older POP servers will *lock* the mailbox. Locking might mean that no alterations to the mailbox may be made, or even that no more mail may be delivered until the lock is gone. The problem is that some POP servers do not properly detect errors, and will keep a box locked indefinitely if your connection gets hung up without your calling `quit()`. At one time, the world's most popular POP server fell into this category!

So it is vital to always call `quit()` in your Python programs when finishing up a POP session. You will note that all of the program listings shown here are careful to always `quit()` down in a `finally` block that Python is guaranteed to execute last.

Obtaining Mailbox Information

The preceding example showed you `stat()`, which returns the number of messages in the mailbox and their total size. Another useful POP command is `list()`, which returns more detailed information about each message.

The most interesting part is the message number, which is required to retrieve messages later. Note that there may be gaps in message numbers: a mailbox may, for example, contain message numbers 1, 2, 5, 6, and 9. Also, the number assigned to a particular message may be different on each connection you make to the POP server.

Listing 14–3 shows how to use the `list()` command to display information about each message.

Listing 14–3. *Using the POP `list()` Command*

```
#!/usr/bin/env python
# POP mailbox scanning - Chapter 14 - mailbox.py

import getpass, poplib, sys

if len(sys.argv) != 3:
»   print 'usage: %s hostname user' % sys.argv[0]
»   exit(2)

hostname, user = sys.argv[1:]
passwd = getpass.getpass()

p = poplib.POP3_SSL(hostname)
try:
»   p.user(user)
»   p.pass_(passwd)
except poplib.error_proto, e:
»   print "Login failed:", e
else:
»   response, listings, octet_count = p.list()
»   for listing in listings:
»   »   number, size = listing.split()
»   »   print "Message %s has %s bytes" % (number, size)
```

```
finally:
»   p.quit()
```

The `list()` function returns a tuple containing three items; you should generally pay attention to the second item. Here is its raw output for one of my POP mailboxes at the moment, which has three messages in it:

```
('+OK 3 messages (5675 bytes)', ['1 2395', '2 1626',
 '3 1654'], 24)
```

The three strings inside the second item give the message number and size for each of the three messages in my in-box. The simple parsing performed by Listing 14–3 lets it present the output in a prettier format:

```
$ ./mailbox.py popserver.example.com testuser
Password:
Message 1 has 2395 bytes
Message 2 has 1626 bytes
Message 3 has 1654 bytes
```

Downloading and Deleting Messages

You should now be getting the hang of POP: when using `poplib` you get to issue small atomic commands that always return a tuple inside which are various strings and lists of strings showing you the result. We are now ready to actually manipulate messages! The three relevant methods, which all identify messages using the same integer identifiers that are returned by `list()`, are these:

- `retr(num)`: This method downloads a single message and returns a tuple containing a result code and the message itself, delivered as a list of lines. This will cause most POP servers to set the "seen" flag for the message to "true," barring you from ever seeing it from POP again (unless you have another way into your mailbox that lets you set messages back to "Unread").
- `top(num, body_lines)`:This method returns its result in the same format as `retr()` *without* marking the message as "seen." But instead of returning the whole message, it just returns the headers plus however many lines of the body you ask for in `body_lines`. This is useful for previewing messages if you want to let the user decide which ones to download.
- `dele(num)`: This method marks the message for deletion from the POP server, to take place when you quit this POP session. Typically you would do this only if the user directly requests irrevocable destruction of the message, or if you have stored the message to disk and used something like `fsync()` to assure the data's safety.

To put everything together, take a look at Listing 14–4, which is a fairly functional e-mail client that speaks POP! It checks your in-box to determine how many messages there are and to learn what their numbers are; then it uses `top()` to offer a preview of each one; and, at the user's option, it can retrieve the whole message, and can also delete it from the mailbox.

Listing 14–4. *A Simple POP E-mail Reader*

```
#!/usr/bin/env python
# POP mailbox downloader with deletion - Chapter 14
# download-and-delete.py

import email, getpass, poplib, sys
```

```
if len(sys.argv) != 3:
»   print 'usage: %s hostname user' % sys.argv[0]
»   exit(2)
hostname, user = sys.argv[1:]

passwd = getpass.getpass()

p = poplib.POP3_SSL(hostname)
try:
»   p.user(user)
»   p.pass_(passwd)
except poplib.error_proto, e:
»   print "Login failed:", e
else:
»   response, listings, octets = p.list()
»   for listing in listings:
»   »   number, size = listing.split()
»   »   print 'Message', number, '(size is', size, 'bytes):'
»   »   print
»   »   response, lines, octets = p.top(number, 0)
»   »   message = email.message_from_string('\n'.join(lines))
»   »   for header in 'From', 'To', 'Subject', 'Date':
»   »   »   if header in message:
»   »   »   »   print header + ':', message[header]
»   »   print
»   »   print 'Read this message [ny]?'
»   »   answer = raw_input()
»   »   if answer.lower().startswith('y'):
»   »   »   response, lines, octets = p.retr(number)
»   »   »   message = email.message_from_string('\n'.join(lines))
»   »   »   print '-' * 72
»   »   »   for part in message.walk():
»   »   »   »   if part.get_content_type() == 'text/plain':
»   »   »   »   »   print part.get_payload()
»   »   »   »   »   print '-' * 72
»   »   print
»   »   print 'Delete this message [ny]?'
»   »   answer = raw_input()
»   »   if answer.lower().startswith('y'):
»   »   »   p.dele(number)
»   »   »   print 'Deleted.'
finally:
»   p.quit()
```

You will note that the listing uses the `email` module, introduced in Chapter 12, to great advantage, since even fancy modern MIME e-mails with HTML and images usually have a `text/plain` section that a simple program like this can print to the screen.

If you run this program, you'll see output similar to this:

```
$ ./download-and-delete.py pop.gmail.com my_gmail_acct
Message 1 (size is 1847 bytes):
From: root@server.example.com
To: Brandon Rhodes <brandon.craig.rhodes@gmail.com>
```

```
Subject: Backup complete
Date: Tue, 13 Apr 2010 16:56:43 -0700 (PDT)
Read this message [ny]?
n
Delete this message [ny]?
y
Deleted.
```

Summary

POP, the Post Office Protocol, provides a simple way to download e-mail messages stored on a remote server. With Python's `poplib` interface, you can obtain information about the number of messages in a mailbox and the size of each message. You can also retrieve or delete individual messages by number.

Connecting to a POP server may lock a mailbox. Therefore, it's important to try to keep POP sessions as brief as possible and always call `quit()` when done.

POP should be used with SSL when possible to protect your passwords and e-mail message contents. In the absence of SSL, try to at least use APOP, and send your password in the clear only in dire circumstances where you desperately need to POP and none of the fancier options work.

Although POP is a simple and widely deployed protocol, it has a number of drawbacks that make it unsuitable for some applications. For instance, it can access only one folder, and does not provide persistent tracking of individual messages. The next chapter discusses IMAP, a protocol that provides the features of POP with a number of new features as well.

CHAPTER 15

IMAP

At first glance, the Internet Message Access Protocol (IMAP) resembles the POP protocol described in Chapter 14. And if you have read the first sections of Chapter 13, which give the whole picture of how e-mail travels across the Internet, you will already know that the two protocols fill a quite similar role: POP and IMAP are two ways that a laptop or desktop computer can connect to a larger Internet server to view and manipulate a user's e-mail.

But there the resemblance ends. Whereas the capabilities of POP are rather anemic—the user can download new messages to his or her personal computer—the IMAP protocol offers such a full array of capabilities that many users store their e-mail permanently on the server, keeping it safe from a laptop or desktop hard drive crash. Among the advantages that IMAP has over POP are the following:

- Mail can be sorted into several folders, rather than having to arrive in a single in-box.
- Flags are supported for each message, like "read," "replied," "seen," and "deleted."
- Messages can be searched for text strings right on the server, without having to download each one.
- A locally stored message can be uploaded directly to one of the remove folders.
- Persistent unique message numbers are maintained, making robust synchronization possible between a local message store and the messages kept on the server.
- Folders can be shared with other users, or marked read-only.
- Some IMAP servers can present non-mail sources, like Usenet newsgroups, as though they were mail folders.
- An IMAP client can selectively download one part of a message—for example, grabbing a particular attachment, or only the message headers, without having to wait to download the rest of the message.

These features, taken together, mean that IMAP can be used for many more operations than the simple download-and-delete spasm that POP supports. Many mail readers, like Thunderbird and Outlook, can present IMAP folders so they operate with the same capabilities of locally stored folders. When a user clicks a message, the mail reader downloads it from the IMAP server and displays it, instead of having to download all of the messages in advance; the reader can also set the message's "read" flag at the same time.

THE IMAP PROTOCOL

Purpose: **Read, arrange, and delete mail from mail folders**

Standard: **RFC 3501 (2003)**

Runs atop: **TCP/IP**

Default port: **143 (cleartext), 993 (SSL)**

Library: imaplib, IMAPClient

Exceptions: `socket.error, socket.gaierror, IMAP4.error, IMAP4.abort, IMAP4.readonly`

IMAP clients can also synchronize themselves with an IMAP server. Someone about to leave on a business trip might download an IMAP folder to a laptop. Then, on the road, mail might be read, deleted, or replied to; the user's mail program would record these actions. When the laptop finally reconnects to the network, their e-mail client can mark the messages on the server with the same "read" or "replied" flags already set locally, and can even go ahead and delete the messages from the server that were already deleted locally so that the user does not see them twice.

The result is one of IMAP's biggest advantages over POP: users can see the same mail, in the same state, from all of their laptop and desktop machines. Either the poor POP users must, instead, see the same mail multiple times (if they tell their e-mail clients to leave mail on the server), or each message will be downloaded only once to the machine on which they happen to read it (if the e-mail clients delete the mail), which means that their mail winds up scattered across all of the machines from which they check it. IMAP users avoid this dilemma.

Of course, IMAP can also be used in exactly the same manner as POP—to download mail, store it locally, and delete the messages immediately from the server—for those who do not want or need its advanced features.

There are several versions of the IMAP protocol available. The most recent, and by far the most popular, is known as IMAP4rev1; in fact, the term "IMAP" is today generally synonymous with IMAP4rev1. This chapter assumes that IMAP servers are IMAP4rev1 servers. Very old IMAP servers, which are quite uncommon, may not support all features discussed in this chapter.

There is also a good how-to about writing an IMAP client at the following links:

```
http://www.dovecot.org/imap-client-coding-howto.html
http://www.imapwiki.org/ClientImplementation
```

If you are doing anything beyond simply writing a small single-purpose client to summarize the messages in your in-box or automatically download attachments, then you should read the foregoing resources thoroughly—or a book on IMAP, if you want a more thorough reference—so that you can handle correctly all of the situations you might run into with different servers and their implementations of IMAP. This chapter will teach just the basics, with a focus on how to best connect from Python.

Understanding IMAP in Python

The Python Standard Library contains an IMAP client interface named `imaplib`, which does offer rudimentary access to the protocol. Unfortunately, it limits itself to knowing how to send requests and deliver their responses back to your code. It makes no attempt to actually implement the detailed rules in the IMAP specification for parsing the returned data.

As an example of how values returned from `imaplib` are usually too raw to be usefully used in a program, take a look at Listing 15–1. It is a simple script that uses `imaplib` to connect to an IMAP account, list the "capabilities" that the server advertises, and then display the status code and data returned by the LIST command.

Listing 15–1. *Connecting to IMAP and Listing Folders*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 15 - open_imaplib.py
# Opening an IMAP connection with the pitiful Python Standard Library

import getpass, imaplib, sys

try:
»   hostname, username = sys.argv[1:]
except ValueError:
»   print 'usage: %s hostname username' % sys.argv[0]
»   sys.exit(2)

m = imaplib.IMAP4_SSL(hostname)
m.login(username, getpass.getpass())
print 'Capabilities:', m.capabilities
print 'Listing mailboxes '
status, data = m.list()
print 'Status:', repr(status)
print 'Data:'
for datum in data:
»   print repr(datum)
m.logout()
```

If you run this script with appropriate arguments, it will start by asking for your password—IMAP authentication is almost always accomplished through a username and password:

```
$ python open_imaplib.py imap.example.com brandon@example.com
Password:
```

If your password is correct, it will then print out a response that looks something like the result shown in Listing 15–2. As promised, we see first the "capabilities," which list the IMAP features that this server supports. And, we must admit, the type of this list is very Pythonic: whatever form the list had on the wire has been turned into a pleasant tuple of strings.

Listing 15–2. *Example Output of the Previous Listing*

```
Capabilities: ('IMAP4REV1', 'UNSELECT', 'IDLE', 'NAMESPACE', 'QUOTA',
 'XLIST', 'CHILDREN', 'XYZZY', 'SASL-IR', 'AUTH=XOAUTH')
Listing mailboxes
Status: 'OK'
Data:
'(\\HasNoChildren) "/" "INBOX"'
'(\\HasNoChildren) "/" "Personal"'
'(\\HasNoChildren) "/" "Receipts"'
'(\\HasNoChildren) "/" "Travel"'
'(\\HasNoChildren) "/" "Work"'
'(\\Noselect \\HasChildren) "/" "[Gmail]"'
'(\\HasChildren \\HasNoChildren) "/" "[Gmail]/All Mail"'
```

```
'(\\HasNoChildren) "/" "[Gmail]/Drafts"'
'(\\HasChildren \\HasNoChildren) "/" "[Gmail]/Sent Mail"'
'(\\HasNoChildren) "/" "[Gmail]/Spam"'
'(\\HasNoChildren) "/" "[Gmail]/Starred"'
'(\\HasChildren \\HasNoChildren) "/" "[Gmail]/Trash"'
```

But things fall apart when we turn to the result of the `list()` method. First, we have been returned its status code manually, and code that uses `imaplib` has to incessantly check for whether the code is `'OK'` or whether it indicates an error. This is not terribly Pythonic, since usually Python programs can run along without doing error checking and be secure in the knowledge that an exception will be thrown if anything goes wrong.

Second, `imaplib` gives us no help in interpreting the results! The list of e-mail folders in this IMAP account uses all sorts of protocol-specific quoting: each item in the list names the flags set on each folder, then designates the character used to separate folders and sub-folders (the slash character, in this case), and then finally supplies the quoted name of the folder. But all of this is returned to us raw, leaving it to us to interpret strings like the following:

```
(\HasChildren \HasNoChildren) "/" "[Gmail]/Sent Mail"
```

So unless you want to implement several details of the protocol yourself, you will want a more capable IMAP client library.

IMAPClient

Fortunately, a popular and battle-tested IMAP library for Python does exist, and is available for easy installation from the Python Package Index. The `IMAPClient` package is written by a friendly Python programmer named Menno Smits, and in fact uses the Standard Library `imaplib` behind the scenes to do its work.

If you want to try out `IMAPClient`, try installing it in a "virtualenv," as described in Chapter 1. Once installed, you can use the `python` interpreter in the virtual environment to run the program shown in Listing 15–3.

Listing 15–3. *Listing IMAP Folders with IMAPClient*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 15 - open_imap.py
# Opening an IMAP connection with the powerful IMAPClient

import getpass, sys
from imapclient import IMAPClient

try:
»   hostname, username = sys.argv[1:]
except ValueError:
»   print 'usage: %s hostname username' % sys.argv[0]
»   sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
»   c.login(username, getpass.getpass())
except c.Error, e:
»   print 'Could not log in:', e
»   sys.exit(1)

print 'Capabilities:', c.capabilities()
```

```
print 'Listing mailboxes:'
data = c.list_folders()
for flags, delimiter, folder_name in data:
»   print '  %-30s%s %s' % (' '.join(flags), delimiter, folder_name)
c.logout()
```

You can see immediately from the code that more details of the protocol exchange are now being handled on our behalf. For example, we no longer get a status code back that we have to check every time we run a command; instead, the library is doing that check for us and will raise an exception to stop us in our tracks if anything goes wrong.

Second, you can see that each result from the LIST command—which in this library is offered as the `list_folders()` method instead of the `list()` method offered by `imaplib`—has already been parsed into Python data types for us. Each line of data comes back as a tuple giving us the folder flags, folder name delimiter, and folder name, and the flags themselves are a sequence of strings.

Take a look at Listing 15–4 for what the output of this second script looks like.

Listing 15–4. *Properly Parsed Flags and Folder Names*

```
Capabilities: ('IMAP4REV1', 'UNSELECT', 'IDLE', 'NAMESPACE', 'QUOTA', 'XLIST', 'CHILDREN',
'XYZZY', 'SASL-IR', 'AUTH=XOAUTH')
Listing mailboxes:
  \HasNoChildren                / INBOX
  \HasNoChildren                / Personal
  \HasNoChildren                / Receipts
  \HasNoChildren                / Travel
  \HasNoChildren                / Work
  \Noselect \HasChildren        / [Gmail]
  \HasChildren \HasNoChildren   / [Gmail]/All Mail
  \HasNoChildren                / [Gmail]/Drafts
  \HasChildren \HasNoChildren   / [Gmail]/Sent Mail
  \HasNoChildren                / [Gmail]/Spam
  \HasNoChildren                / [Gmail]/Starred
  \HasChildren \HasNoChildren   / [Gmail]/Trash
```

The standard flags listed for each folder may be zero or more of the following:

- `\Noinferiors`: This means that the folder does not contain any sub-folders and that it is not possible for it to contain sub-folders in the future. Your IMAP client will receive an error if it tries to create a sub-folder under this folder.

- `\Noselect`: This means that it is not possible to run `select_folder()` on this folder—that is, this folder does not and cannot contain any messages. (Perhaps it exists just to allow sub-folders beneath it, as one possibility.)

- `\Marked`: This means that the server considers this box to be interesting in some way; generally, this indicates that new messages have been delivered since the last time the folder was selected. However, the absence of `\Marked` does *not* guarantee that the folder does not contain new messages; some servers simply do not implement `\Marked` at all.

- `\Unmarked`: This guarantees that the folder doesn't contain new messages.

Some servers return additional flags not covered in the standard. Your code must be able to accept and ignore those additional flags.

Examining Folders

Before you can actually download, search, or modify any messages, you must "select" a particular folder to look at. This means that the IMAP protocol is stateful: it remembers which folder you are currently looking at, and its commands operate on the current folder without making you repeat its name over and over again. This can make interaction more pleasant, but it also means that your program has to be careful that it always knows what folder is selected or it might wind up doing something to the wrong folder.

So when you "select" a folder, you tell the IMAP server that all the following commands—until you change folders, or exit the current one—will apply to the selected folder.

When selecting, you have the option to select the folder "read only" by supplying a `readonly=True` argument. This causes any operations that would delete or modify messages to return an error message should you attempt them. Besides preventing you from making any mistakes when you meant to leave all of the messages intact, the fact that you are just reading can be used by the server to optimize access to the folder (for example, it might read-lock but not write-lock the actual folder storage on disk while you have it selected).

Message Numbers vs. UIDs

IMAP provides two different ways to refer to a specific message within a folder: by a temporary message number (which typically goes 1, 2, 3, and so forth) or by a UID (unique identifier). The difference between the two lies with persistence. Message numbers are assigned right when you select the folder. This means they can be pretty and sequential, but it also means that if you revisit the same folder later, then a given message may have a different number. For programs such as live mail readers or simple download scripts, this behavior (which is the same as POP) is fine; you do not need the numbers to stay the same.

But a UID, by contrast, is designed to remain the same even if you close your connection to the server and do not reconnect again for another week. If a message had UID 1053 today, then the same message will have UID 1053 tomorrow, and no other message in that folder will ever have UID 1053. If you are writing a synchronization tool, this behavior is quite useful! It will allow you to verify with 100% percent certainty that actions are being taken against the correct message. This is one of the things that make IMAP so much more fun than POP.

Note that if you return to an IMAP account and the user has—without telling you—deleted a folder and then created a new one with the same name, then it might look to your program as though the same folder is present but that the UID numbers are conflicting and no longer agree. Even a folder re-name, if you fail to notice it, might make you lose track of which messages in the IMAP account correspond to which messages you have already downloaded. But it turns out that IMAP is prepared to protect you against this, and (as we will see soon) provides a `UIDVALIDITY` folder attribute that you can compare from one session to the next to see whether UIDs in the folder will really correspond to the UIDs that the same messages had when you last connected.

Most IMAP commands that work with specific messages can take either message numbers or UIDs. Normally, `IMAPClient` always uses UIDs and ignores the temporary message numbers assigned by IMAP. But if you want to see the temporary numbers instead, simply instantiate `IMAPClient` with a `use_uid=False` argument—or, you can even set the value of the class's `use_uid` attribute to `False` and `True` on the fly during your IMAP session.

Message Ranges

Most IMAP commands that work with messages can work with one or more messages. This can make processing far faster if you need a whole group of messages. Instead of issuing separate commands and receiving separate responses for each individual message, you can operate on a group of messages as a whole. The operation works faster since you no longer have to deal with a network round-trip for every single command.

When you supply a message number, you can instead supply a comma-separated list of message numbers. And, if you want all messages whose numbers are in a range but you do not want to have to list all of their numbers (or if you do not even know their numbers—maybe you want "everything starting with message one" without having to fetch their numbers first), you can use a colon to separate the start and end message numbers. An asterisk means "and all of the rest of the messages." Here is an example specification:

```
2,4:6,20:*
```

It means "message 2," "messages 4 through 6," and "message 20 through the end of the mail folder."

Summary Information

When you first select a folder, the IMAP server provides some summary information about it—about the folder itself and also about its messages.

The summary is returned by `IMAPClient` as a dictionary. Here are the keys that most IMAP servers will return when you run `select_folder()`:

- `EXISTS`: An integer giving the number of messages in the folder
- `FLAGS`: A list of the flags that can be set on messages in this folder
- `RECENT`: Specifies the server's approximation of the number of messages that have appeared in the folder since the last time an IMAP client ran `select_folder()` on it.
- `PERMANENTFLAGS`: Specifies the list of custom flags that can be set on messages; this is usually empty.
- `UIDNEXT`: The server's guess about the UID that will be assigned to the next incoming (or uploaded) message
- `UIDVALIDITY`: A string that can be used by clients to verify that the UID numbering has not changed; if you come back to a folder and this is a different value than the last time you connected, then the UID number has started over and your stored UID values are no longer valid.
- `UNSEEN`: Specifies the message number of the first unseen message (one without the `\Seen` flag) in the folder

Of these flags, servers are only required to return `FLAGS`, `EXISTS`, and `RECENT`, though most will include at least `UIDVALIDITY` as well. Listing 15–5 shows an example program that reads and displays the summary information of my `INBOX` mail folder.

Listing 15–5. *Displaying Folder Summary Information*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 15 - folder_info.py
# Opening an IMAP connection with IMAPClient and listing folder information.

import getpass, sys
from imapclient import IMAPClient

try:
»   hostname, username = sys.argv[1:]
except ValueError:
»   print 'usage: %s hostname username' % sys.argv[0]
»   sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
»   c.login(username, getpass.getpass())
except c.Error, e:
»   print 'Could not log in:', e
»   sys.exit(1)
else:
»   select_dict = c.select_folder('INBOX', readonly=True)
»   for k, v in select_dict.items():
»   »   print '%s: %r' % (k, v)
»   c.logout()
```

When run, this program displays results such as this:

```
$ ./folder_info.py imap.example.com brandon@example.com
Password:
EXISTS: 3
PERMANENTFLAGS: ('\\Answered', '\\Flagged', '\\Draft', '\\Deleted',
»   »   »   »   '\\Seen', '\\*')
READ-WRITE: True
UIDNEXT: 2626
FLAGS: ('\\Answered', '\\Flagged', '\\Draft', '\\Deleted', '\\Seen')
UIDVALIDITY: 1
RECENT: 0
```

That shows that my INBOX folder contains three messages, none of which have arrived since I last checked. If your program is interested in using UIDs that it stored during previous sessions, remember to compare the UIDVALIDITY to a stored value from a previous session.

Downloading an Entire Mailbox

With IMAP, the FETCH command is used to download mail, which IMAPClient exposes as its fetch() method.

The simplest way to fetch involves downloading all messages at once, in a single big gulp. While this is simplest and requires the least network traffic (since you do not have to issue repeated commands and receive multiple responses), it does mean that all of the returned messages will need to sit in memory

together as your program examines them. For very large mailboxes whose messages have lots of attachments, this is obviously not practical!

Listing 15–6 downloads all of the messages from my INBOX folder into your computer's memory in a Python data structure, and then displays a bit of summary information about each one.

Listing 15–6. *Downloading All Messages in a Folder*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 15 - mailbox_summary.py
# Opening an IMAP connection with IMAPClient and retrieving mailbox messages.

import email, getpass, sys
from imapclient import IMAPClient

try:
»   hostname, username, foldername = sys.argv[1:]
except ValueError:
»   print 'usage: %s hostname username folder' % sys.argv[0]
»   sys.exit(2)

c = IMAPClient(hostname, ssl=True)
try:
»   c.login(username, getpass.getpass())
except c.Error, e:
»   print 'Could not log in:', e
»   sys.exit(1)

c.select_folder(foldername, readonly=True)
msgdict = c.fetch('1:*', ['BODY.PEEK[]'])
for message_id, message in msgdict.items():
»   e = email.message_from_string(message['BODY[]'])
»   print message_id, e['From']
»   payload = e.get_payload()
»   if isinstance(payload, list):
»   »   part_content_types = [ part.get_content_type() for part in payload ]
»   »   print '  Parts:', ' '.join(part_content_types)
»   else:
»   »   print '  ', ' '.join(payload[:60].split()), '...'
c.logout()
```

Remember that IMAP is stateful: first we use `select_folder()` to put us "inside" the given folder, and then we can run `fetch()` to ask for message content. (You can later run `close_folder()` if you want to leave and not be inside a given folder any more.) The range `'1:*'` means "the first message through the end of the mail folder," because message IDs—whether temporary or UIDs—are always positive integers.

The perhaps odd-looking string `'BODY.PEEK[]'` is the way to ask IMAP for the "whole body" of the message. The string `'BODY[]'` means "the whole message"; inside the square brackets, as we will see, you can instead ask for just specific parts of a message.

And `PEEK` indicates that you are just looking inside the message to build a summary, and that you do *not* want the server to automatically set the `\Seen` flag on all of these messages for you and thus ruin its memory of which messages the user has read. (This seemed a nice feature for me to add to a little script like this that you might run against a real mailbox—I would not want to mark all your messages as read!)

The dictionary that is returned maps message UIDs to dictionaries giving information about each message. As we iterate across its keys and values, we look in each message-dictionary for the `'BODY[]'`

key that IMAP has filled in with the information about the message that we asked for: its full text, returned as a large string.

Using the email module that we learned about in Chapter 12, the script asks Python to grab the From: line and a bit of the message's content, and print them to the screen as a summary. Of course, if you wanted to extend this script so that you save the messages in a file or database instead, you can just omit the email parsing step and instead treat the message body as a single string to be deposited in storage and parsed later.

Here is what it looks like to run this script:

```
$ ./mailbox_summary.py imap.example.com brandon INBOX
Password:
2590 "Amazon.com" <order-update@amazon.com>
  Dear Brandon, Portable Power Systems, Inc. shipped the follo ...
2469 Meetup Reminder <info@meetup.com>
  Parts: text/plain text/html
2470 billing@linode.com
  Thank you. Please note that charges will appear as "Linode.c ...
```

Of course, if the messages contained large attachments, it could be ruinous to download them in their entirety just to print a summary; but since this is the simplest message-fetching operation, I thought that it would be reasonable to start with it!

Downloading Messages Individually

E-mail messages can be quite large, and so can mail folders—many mail systems permit users to have hundreds or thousands of messages, that can each be 10MB or more. That kind of mailbox can easily exceed the RAM on the client machine if its contents are all downloaded at once, as in the previous example.

To help network-based mail clients that do not want to keep local copies of every message, IMAP supports several operations besides the big "fetch the whole message" command that we saw in the previous section.

- An e-mail's headers can be downloaded as a block of text, separately from the message.
- Particular headers from a message can be requested and returned.
- The server can be asked to recursively explore and return an outline of the MIME structure of a message.
- The text of particular sections of the message can be returned.

This allows IMAP clients to perform very efficient queries that download only the information they need to display for the user, decreasing the load on the IMAP server and the network, and allowing results to be displayed more quickly to the user.

For an example of how a simple IMAP client works, examine Listing 15–7, which puts together a number of ideas about browsing an IMAP account. Hopefully this provides more context than would be possible if these features were spread out over a half-dozen shorter program listings at this point in the chapter! You can see that the client consists of three concentric loops that each take input from the user as he or she views the list of mail folders, then the list of messages within a particular mail folder, and finally the sections of a specific message.

Listing 15–7. *A Simple IMAP Client*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 15 - simple_client.py
# Letting a user browse folders, messages, and message parts.

import getpass, sys
from imapclient import IMAPClient
try:
»   hostname, username = sys.argv[1:]
except ValueError:
»   print 'usage: %s hostname username' % sys.argv[0]
»   sys.exit(2)

banner = '-' * 72

c = IMAPClient(hostname, ssl=True)
try:
»   c.login(username, getpass.getpass())
except c.Error, e:
»   print 'Could not log in:', e
»   sys.exit(1)

def display_structure(structure, parentparts=[]):
»   """Attractively display a given message structure."""
»   # The whole body of the message is named 'TEXT'.
»   if parentparts:
»   »   name = '.'.join(parentparts)
»   else:
»   »   print 'HEADER'
»   »   name = 'TEXT'

»   # Print this part's designation and its MIME type.
»   is_multipart = isinstance(structure[0], list)
»   if is_multipart:
»   »   parttype = 'multipart/%s' % structure[1].lower()
»   else:
»   »   parttype = ('%s/%s' % structure[:2]).lower()
»   print '%-9s' % name, parttype,
»   # For a multipart part, print all of its subordinate parts; for
»   # other parts, print their disposition (if available).
»   if is_multipart:
»   »   print
»   »   subparts = structure[0]
»   »   for i in range(len(subparts)):
»   »   »   display_structure(subparts[i], parentparts + [ str(i + 1) ])
»   else:
»   »   if structure[6]:
»   »   »   print 'size=%s' % structure[6],
»   »   if structure[8]:
»   »   »   disposition, namevalues = structure[8]
»   »   »   print disposition,
»   »   »   for i in range(0, len(namevalues), 2):
»   »   »   »   print '%s=%r' % namevalues[i:i+2]
```

```
        print

def explore_message(c, uid):
    """Let the user view various parts of a given message."""
    msgdict = c.fetch(uid, ['BODYSTRUCTURE', 'FLAGS'])

    while True:
        print
        print 'Flags:',
        flaglist = msgdict[uid]['FLAGS']
        if flaglist:
            print ' '.join(flaglist)
        else:
            print 'none'
        display_structure(msgdict[uid]['BODYSTRUCTURE'])
        print
        reply = raw_input('Message %s - type a part name, or "q" to quit: '
                                    % uid).strip()
        print
        if reply.lower().startswith('q'):
            break
        key = 'BODY[%s]' % reply
        try:
            msgdict2 = c.fetch(uid, [key])
        except c._imap.error:
            print 'Error - cannot fetch section %r' % reply
        else:
            content = msgdict2[uid][key]
            if content:
                print banner
                print content.strip()
                print banner
            else:
                print '(No such section)'

def explore_folder(c, name):
    """List the messages in folder `name` and let the user choose one."""

    while True:
        c.select_folder(name, readonly=True)
        msgdict = c.fetch('1:*', ['BODY.PEEK[HEADER.FIELDS (FROM SUBJECT)]',
                                            'FLAGS', 'INTERNALDATE', 'RFC822.SIZE'])
        print
        for uid in sorted(msgdict):
            items = msgdict[uid]
            print '%6d  %20s  %6d bytes  %s' % (
                uid, items['INTERNALDATE'], items['RFC822.SIZE'],
                ' '.join(items['FLAGS']))
            for i in items['BODY[HEADER.FIELDS (FROM SUBJECT)]'].splitlines():
                print ' ' * 6, i.strip()

        reply = raw_input('Folder %s - type a message UID, or "q" to quit: '
                                    % name).strip()
        if reply.lower().startswith('q'):
```

```
            break
        try:
            reply = int(reply)
        except ValueError:
            print 'Please type an integer or "q" to quit'
        else:
            if reply in msgdict:
                explore_message(c, reply)

    c.close_folder()

def explore_account(c):
    """Display the folders in this IMAP account and let the user choose one."""

    while True:

        print
        folderflags = {}
        data = c.list_folders()
        for flags, delimiter, name in data:
            folderflags[name] = flags
        for name in sorted(folderflags.keys()):
            print '%-30s %s' % (name, ' '.join(folderflags[name]))
        print

        reply = raw_input('Type a folder name, or "q" to quit: ').strip()
        if reply.lower().startswith('q'):
            break
        if reply in folderflags:
            explore_folder(c, reply)
        else:
            print 'Error: no folder named', repr(reply)

if __name__ == '__main__':
    explore_account(c)
```

You can see that the outer function uses a simple `list_folders()` call to present the user with a list of his or her mail folders, like some of the program listings we have seen already. Each folder's IMAP flags are also displayed. This lets the program give the user a choice between folders:

```
INBOX                          \HasNoChildren
Receipts                       \HasNoChildren
Travel                         \HasNoChildren
Work                           \HasNoChildren
Type a folder name, or "q" to quit:
```

Once a user has selected a folder, things become more interesting: a summary has to be printed for each message. Different e-mail clients make different choices about what information to present about each message in a folder; Listing 15–7 chooses to select a few header fields together with the message's date and size. Note that it is careful to use `BODY.PEEK` instead of `BODY` to fetch these items, since the IMAP server would otherwise mark the messages as `\Seen` merely because they had been displayed in a summary!

The results of this `fetch()` call are printed to the screen once an e-mail folder has been selected:

```
2703   2010-09-28 21:32:13   19129 bytes  \Seen
```

```
»     From: Brandon Craig Rhodes
»     Subject: Digested Articles

2704   2010-09-28 23:03:45   15354 bytes
»     Subject: Re: [venv] Building a virtual environment for offline testing
»     From: "W. Craig Trader"

2705   2010-09-29 08:11:38   10694 bytes
»     Subject: Re: [venv] Building a virtual environment for offline testing
»     From: Hugo Lopes Tavares

Folder INBOX - type a message UID, or "q" to quit:
```

As you can see, the fact that several items of interest can be supplied to the IMAP `fetch()` command lets us build fairly sophisticated message summaries with only a single round-trip to the server!

Once the user has selected a particular message, we use a technique that we have not discussed so far: we ask `fetch()` to return the `BODYSTRUCTURE` of the message, which is the key to seeing a MIME message's parts without having to download its entire text. Instead of making us pull several megabytes over the network just to list a large message's attachments, `BODYSTRUCTURE` simply lists its MIME sections as a recursive data structure.

Simple MIME parts are returned as a tuple:

```
('TEXT', 'PLAIN', ('CHARSET', 'US-ASCII'), None, None, '7BIT', 2279, 48)
```

The elements of this tuple, which are detailed in section 7.4.2 of RFC 3501, are as follows (starting from item index zero, of course):

1. MIME type
2. MIME subtype
3. Body parameters, presented as a tuple (`name value name value ...`) where each parameter name is followed by its value
4. Content ID
5. Content description
6. Content encoding
7. Content size, in bytesva
8. For textual MIME types, this gives the content length in lines.

When the IMAP server sees that a message is multipart, or when it examines one of the parts of the message that it discovers is itself multipart (see Chapter 12 for more information about how MIME messages can nest other MIME messages inside them), then the tuple it returns will begin with a list of sub-structures, which are each a tuple laid out just like the outer structure. Then it will finish with some information about the multipart container that bound those sections together:

```
([(...), (...)], "MIXED", ('BOUNDARY', '=-=-='), None, None)
```

The value `"MIXED"` indicates exactly what kind of multipart container is being represented—in this case, the full type is `multipart/mixed`. Other common “multipart” subtypes besides “mixed” are alternative, digest, and parallel. The remaining items beyond the multipart type are optional, but if present, provide a set of name-value parameters (here indicating what the MIME multipart boundary string was), the multipart's disposition, its language, and its location (typically given by a URL).

Given these rules, you can see how a recursive routine like `display_structure()` in Listing 15–7 is perfect for unwinding and displaying the hierarchy of parts in a message. When the IMAP server returns a `BODYSTRUCTURE`, the routine goes to work and prints out something like this for examination by the user:

```
Folder INBOX - type a message UID, or "q" to quit: 2701
Flags: \Seen
HEADER
TEXT      multipart/mixed
1         multipart/alternative
1.1       text/plain size=253
1.2       text/html size=508
2         application/octet-stream size=5448 ATTACHMENT FILENAME='test.py'
Message 2701 - type a part name, or "q" to quit:
```

You can see that the message whose structure is shown here is a quite typical modern e-mail, with a fancy rich-text HTML portion for users who view it in a browser or modern e-mail client, and a plain-text version of the same message for people using more traditional devices or applications. It also contains a file attachment, complete with a suggested file name in case the user wants to download it to the local filesystem. Our sample program does not attempt to save anything to the hard drive, both for simplicity and safety; instead, the user can select any portion of the message—such as the special sections `HEADER` and `TEXT`, or one of the specific parts like `1.1`—and its content will be printed to the screen.

If you examine the program listing, you will see that all of this is supported simply by calls to the IMAP `fetch()` method. Part names like `HEADER` and `1.1` are simply more options for what you can specify when you call `fetch()`, and can be used right alongside other values like `BODY.PEEK` and `FLAGS`. The only difference is that the latter values work for all messages, whereas a part name like `2.1.3` would exist only for multipart messages whose structure included a part with that designation.

One oddity you will note is that the IMAP protocol does *not* actually provide you with any of the multipart names that a particular message supports! Instead, you have to count the number of parts listed in the `BODYSTRUCTURE` starting with the index `1` in order to determine which part number you should ask for. You can see that our `display_structure()` routine here uses a simple loop to accomplish this counting.

One final note about the `fetch()` command: it lets you not only pull just the parts of a message that you need at any given moment, but also truncate them in case they are quite long and you just want to provide an excerpt from the beginning to tantalize the user! To use this feature, follow any part name with a slice in angle brackets that indicates what range of characters you want—it works very much like Python's slice operation:

```
BODY[]<0.100>
```

That would return the first 100 bytes of the message body. This can let you inspect both text and the beginning of an attachment to learn more about its content before letting the user decide whether to select or download it.

Flagging and Deleting Messages

You might have noticed, while trying out Listing 15–7 or reading its example output just shown, that IMAP marks messages with attributes called "flags," which typically take the form of a backslash-prefixed word, like `\Seen` for one of the messages just cited. Several of these are standard, and are defined in RFC 3501 for use on all IMAP servers. Here is what the most important ones mean:

- `\Answered`: The user has replied to the message.
- `\Draft`: The user has not finished composing the message.

- \Flagged: The message has somehow been singled out specially; the purpose and meaning of this flag vary between mail readers.
- \Recent: No IMAP client has seen this message before. This flag is unique, in that the flag cannot be added or removed by normal commands; it is automatically removed after the mailbox is selected.
- \Seen: The message has been read.

As you can see, these flags correspond roughly to the information that many mail readers visually present about each message. While the terminology may differ (many clients talk about "new" rather than "not seen" messages), the meaning is broadly understood. Particular servers may also support other flags, and those flags do not necessarily begin with the backslash. Also, the \Recent flag is not reliably supported by all servers, so general-purpose IMAP clients can treat it only as, at best, a hint.

The IMAPClient library supports several methods for working with flags. The simplest retrieves the flags as though you had done a fetch() asking for 'FLAGS', but goes ahead and removes the dictionary around each answer:

```
>>> c.get_flags(2703)
{2703: ('\\Seen',)}
```

There are also calls to add and remove flags from a message:

```
c.remove_flags(2703, ['\\Seen'])
c.add_flags(2703, ['\\Answered'])
```

In case you want to completely change the set of flags for a particular message without figuring out the correct series of adds and removes, you can use set_flags() to unilaterally replace the whole list of message flags with a new one:

```
c.set_flags(2703, ['\\Seen', '\\Answered'])
```

Any of these operations can take a list of message UIDs instead of the single UID shown in these examples.

Deleting Messages

One last interesting use of flags is that it is how IMAP supports message deletion. The process, for safety, takes two steps: first the client marks one or more messages with the \Delete flag; then it calls expunge() to perform the deletions as a single operation.

The IMAPClient library does not make you do this by hand, however (though that would work); instead it hides the fact that flags are involved behind a simple delete_messages() routine that marks the messages for you. It still has to be followed by expunge() if you actually want the operation to take effect, though:

```
c.delete_messages([2703, 2704])
c.expunge()
```

Note that expunge() will reorder the normal IDs of the messages in the mailbox, which is yet another reason for using UIDs instead!

Searching

Searching is another issue that is very important for a protocol designed to let you keep all your mail on the mail server itself: without search, an e-mail client would have to download all of a user's mail anyway the first time he or she wanted to perform a full-text search to find an e-mail message.

The essence of search is simple: you call the `search()` method on an IMAP client instance, and are returned the UIDs (assuming, of course, that you accept the IMAPClient default of `use_uid=True` for your client) of the messages that match your criteria:

```
>>> c.select_folder('INBOX')
>>> c.search('SINCE 20-Aug-2010 TEXT Apress')
[2590L, 2652L, 2653L, 2654L, 2655L, 2699L]
```

These UIDs can then be the subject of a `fetch()` command that retrieves the information about each message that you need in order to present a summary of the search results to the user.

The query shown in the foregoing example combines two criteria, one requesting recent messages (those that have arrived since August 20, 2010, the year that I am typing this) and the other asking that the message text have the word "Apress" somewhere inside, and the result will include only messages that satisfy the first criteria *and* that satisfy the second criteria—that is the result of concatenating two criteria with a space so that they form a single string. If instead you wanted messages that matched just one of the criteria, but not both, you can join them with an `OR` operator:

```
OR (SINCE 20-Aug-2010) (TEXT Apress)
```

There are many criteria that you can combine in order to form a query. Like the rest of IMAP, they are specified in RFC 3501. Some criteria are quite simple, and refer to binary attributes like flags:

```
ALL: Every message in the mailbox
UID (id, ...): Messages with the given UIDs
LARGER n: Messages more than n octets in length
SMALLER m: Messages less than m octets in length
ANSWERED: Have the flag \Answered
DELETED: Have the flag \Deleted
DRAFT: Have the flag \Draft
FLAGGED: Have the flag \Flagged
KEYWORD flag: Have the given keyword flag set
NEW: Have the flag \Recent
OLD: Lack the flag \Recent
UNANSWERED: Lack the flag \Answered
UNDELETED: Lack the flag \Deleted
UNDRAFT: Lack the flag \Draft
UNFLAGGED: Lack the flag \Flagged
UNKEYWORD flag: Lack the given keyword flag
UNSEEN: Lack the flag \Seen
```

There are a number of flags that match items in each message's headers. Each of them searches for a given string in the header of the same name, except for the "send" tests, which look at the `Date` header:

```
BCC string
CC string
FROM string
HEADER name string
SUBJECT string
TO string
```

An IMAP message has two dates: the internal `Date` header specified by the sender, which is called its "send date," and the date at which it actually arrived at the IMAP server. (The former could obviously be a forgery; the latter is as reliable as the IMAP server and its clock.) So there are two sets of criteria for dates, depending on which date you want to query by:

```
BEFORE 01-Jan-1970
ON 01-Jan-1970
SINCE 01-Jan-1970
SENTBEFORE 01-Jan-1970
SENTON 01-Jan-1970
SENTSINCE 01-Jan-1970
```

Finally, there are two search operations that refer to the text of the message itself—these are the big workhorses that support full-text search of the kind your users are probably expecting when they type into a search field in an e-mail client:

```
BODY string: The message body must contain the string.
TEXT string: The entire message, either body or header, must contain the string somewhere.
```

See the documentation for the particular IMAP server you are using to learn whether it returns any "near miss" matches, like those supported by modern search engines, or only exact matches for the words you provide.

If your strings contain any characters that IMAP might consider special, try surrounding them with double-quotes, and then backslash-quote any double-quotes within the strings themselves:

```
>>> c.search(r'TEXT "Quoth the raven, \"Nevermore.\""')
[2652L]
```

Note that by using an `r'...'` string here, I avoided having to double up the backslashes to get single backslashes through to IMAP.

Manipulating Folders and Messages

Creating or deleting folders is done quite simply in IMAP, by providing the name of the folder:

```
c.create_folder('Personal')
c.delete_folder('Work')
```

Some IMAP servers or configurations may not permit these operations, or may have restrictions on naming; be sure to have error checking in place when calling them.

There are two operations that can create new e-mail messages in your IMAP account besides the "normal" means of waiting for people to send them to you.

First, you can copy an existing message from its home folder over into another folder. Start by using `select_folder()` to visit the folder where the messages live, and then run the `copy` method like this:

```
c.select_folder('INBOX')
c.copy([2653L, 2654L], 'TODO')
```

Finally, it is possible to add a message to a mailbox with IMAP. You do not need to send the message first with SMTP; IMAP is all that is needed. Adding a message is a simple process, though there are a couple of things to be aware of.

The primary concern is line endings. Many Unix machines use a single ASCII line feed character (0x0a, or `'\n'` in Python) to designate the end of a line of text. Windows machines use two characters: CR-LF, a carriage return (0x0D, or `'\r'` in Python) followed by a line feed. Older Macs use just the carriage return.

Like many Internet protocols (HTTP comes immediately to mind), IMAP internally uses CR-LF ('\r\n' in Python) to designate the end of a line. Some IMAP servers will have problems if you upload a message that uses any other character for the end of a line. Therefore, you must always be careful to have the correct line endings when you translate uploaded messages. This problem is more common than you might expect, since most local mailbox formats use only '\n' for the end of each line.

However, you must also be cautious in how carefully you change the line endings, because some messages may use '\r\n' somewhere inside despite using only '\n' for the first few dozen lines, and IMAP clients have been known to fail if a message uses both different line endings! The solution is a simple one, thanks to Python's powerful splitlines() string method that recognizes all three possible line endings; simply call the function on your message and then re-join the lines with the standard line ending:

```
>>> 'one\rtwo\nthree\r\nfour'.splitlines()
['one', 'two', 'three', 'four']
>>> '\r\n'.join('one\rtwo\nthree\r\nfour'.splitlines())
'one\r\ntwo\r\nthree\r\nfour'
```

The actual act of appending a message, once you have the line endings correct, is to call the append() method on your IMAP client:

```
c.append('INBOX', my_message)
```

You can also supply a list of flags as a keyword argument, as well as a msg_time to be used as its arrival time by passing a normal Python datetime object.

Asynchrony

Finally, a major admission needs be made about this chapter's approach toward IMAP: even though we have described IMAP as though the protocol were synchronous, it in fact supports clients that want to send dozens of requests down the socket to the server and then receive the answers back in whatever order the server can most efficiently fetch the mail from disk and respond.

The IMAPClient library hides this protocol flexibility by always sending one request, waiting for the response, and then returning that value. But other libraries—and in particular the IMAP capabilities provided inside Twisted Python—let you take advantage of its asynchronicity.

But for most Python programmers needing to script mailbox interactions, the synchronous approach taken in this chapter should work just fine.

Summary

IMAP is a robust protocol for accessing e-mail messages stored on a remote server. Many IMAP libraries exist for Python; imaplib is built into the Standard Library, but requires you to do all sorts of low-level response parsing by yourself. A far better choice is IMAPClient by Menno Smits, which you can install from the Python Package Index.

On an IMAP server, your e-mail messages are grouped into folders, some of which will come pre-defined by your particular IMAP provider and some of which you can create yourself. An IMAP client can create folders, delete folders, insert new messages into a folder, and move existing messages between folders.

Once a folder has been selected, which is the IMAP rough equivalent of a "change directory" command on a filesystem, messages can be listed and fetched very flexibly. Instead of having to download every message in its entirety—though, of course, that is an option—the client can ask for particular information from a message, like a few headers and its message structure, in order to build a

display or summary for the user to click into, pulling message parts and attachments down from the server on demand.

The client can also set flags on each message—some of which are also meaningful to the server—and can delete messages by setting the `\Delete` flag and then performing an expunge operation.

Finally, IMAP offers sophisticated search functionality, again so that common user operations can be supported without requiring the e-mail data to be downloaded to the local machine.

CHAPTER 16

Telnet and SSH

If you have never read it, then you should brew some of your favorite coffee, sit down, and treat yourself to reading Neal Stephenson's essay "In the Beginning Was the Command Line." You can download a copy from his web site, in the form—appropriately enough—of a raw text file:

`http://www.cryptonomicon.com/beginning.html`.

The "command line" is the topic of this chapter: how you can access it over the network, together with enough discussion about its typical behavior to get you through any frustrations you might encounter while trying to use it.

Happily enough, this old-fashioned idea of sending simple textual commands to another computer will, for many readers, be one of the most relevant topics of this book. The main network protocol that we will discuss—SSH, the Secure Shell—seems to be used everywhere to configure and maintain machines of all kinds.

- When you get a new account at a web hosting company like WebFaction and have used their fancy control panel to set up your domain names and list of web applications, the command line is then your primary means of actually installing and running the code behind your web sites.
- Virtual—or physical—servers from companies like Linode, Slicehost, and Rackspace are almost always administered through SSH connections.
- If you build a cloud of dynamically allocated servers using an API-based virtual hosting service like Amazon AWS, you will find that Amazon gives you access to your new host by asking you for an SSH key and installing it so that you can log in to your new instance immediately and without a password.

It is as if, once early computers became able to receive text commands and return text output in response, they reached a kind of pinnacle of usefulness that has never yet been improved upon. Language is the most powerful means humans have for expressing and building meaning, and no amount of pointing, clicking, or dragging with a mouse has ever expressed even a fraction of the nuance that can be communicated when we type—even in the cramped and exacting language of the Unix shell.

Command-Line Automation

Before getting into the details of how the command line works, and how you can access it over the network, we should pause and note that there exist many systems today for automating the entire process. If you have started reading this chapter on programming the networked command line because you have dozens or hundreds of machines to maintain and you need to start sending them all the same commands, then you might find that tools already exist that prevent you from having to read any further—tools that already provide ways to write command scripts, push them out for execution across a

cloud of machines, batch up any error messages or responses for your review, and even save commands in a queue to be re-tried later in case a machine is down and cannot be reached at the moment.

What are your options?

First, the Fabric library is very popular with Python programmers who need to run commands and copy files to remote server machines. As you can see in Listing 16–1, a Fabric script calls very simple functions with names like `put()`, `cd()`, and `run()` to perform operations on the machines to which it connects. We will not cover Fabric in this book, since it does not implement a network protocol of its own, and also because it would be more appropriate in a book on using Python for system administration. But you can learn more about it at its web site: `http://fabfile.org/`.

Although Listing 16–1 is designed to be run by Fabric's own `fab` command-line tool, Fabric can also be used from inside your own Python programs; again, consult their documentation for details.

Listing 16–1. *What Fabric Scripts Look Like*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - fabfile.py
# A sample Fabric script

# Even though this chapter will not cover Fabric, you might want to try
# using Fabric to automate your SSH commands instead of re-inventing the
# wheel.  Here is a script that checks for Python on remote machines.
# Fabric finds this "fabfile.py" automatically if you are in the same
# directory.  Try running both verbosely, and with most messages off:
#
#   $ fab versions:host=server.example.com
#   $ fab --hide=everything versions:host=server.example.com

from fabric.api import *

def versions():
»   with cd('/usr/bin'):
»   »   with settings(hide('warnings'), warn_only=True):
»   »   »   for version in '2.4', '2.5', '2.6', '2.7', '3.0', '3.1':
»   »   »   »   result = run('python%s -c "None"' % version)
»   »   »   »   if not result.failed:
»   »   »   »   »   print "Host", env.host, "has Python", version
```

Another project to check out is Silver Lining, which is being developed by Ian Bicking. It is still very immature, but if you are an experienced programmer who needs its specific capabilities, then you might find that it solves your problems well. This library goes beyond batching commands across many different servers: it will actually create and initialize Ubuntu servers through the "libcloud" Python API, and then install your Python web applications there for you. You can learn more about this promising project at `http://cloudsilverlining.org/`.

Finally, there is "pexpect." While it is not, technically, a program that itself knows how to use the network, it is often used to control the system "ssh" or "telnet" command when a Python programmer wants to automate interactions with a remote prompt of some kind. This typically takes place in a situation where no API for a device is available, and commands simply have to be typed each time the command-line prompt appears. Configuring simple network hardware often requires this kind of clunky step-by-step interaction. You can learn more about "pexpect" here:
`http://pypi.python.org/pypi/pexpect`.

Finally, there are more specific projects that provide mechanisms for remote systems administration. Red Hat and Fedora users might look at `func`, which uses an SSL-encrypted XML-RPC service that lets you write Python programs that perform system configuration and maintenance:
`https://fedorahosted.org/func/`.

But, of course, it might be that no automated solution like these will quite suffice for your project, and you will actually have to roll up your sleeves and learn to manipulate remote-shell protocols yourself. In that case, you have come to the right place; keep reading!

Command-Line Expansion and Quoting

If you have ever typed many commands at a Unix command prompt, you will be aware that not every character you type is interpreted literally. Consider this command, for example (and in this and all following examples in this chapter, I will be using the dollar sign $ as the shell's "prompt" that tells you "it is your turn to type"):

```
$ echo *
Makefile chapter-16.txt formats.ini out.odt source tabify2.py test.py
```

The asterisk * in this command was not interpreted to mean "print out an asterisk character to the screen"; instead, the shell thought I was trying to write a pattern that would match all of the file names in the current directory. To actually print out an asterisk, I have to use another special character—an "escape" character, because it lets me "escape" from the shell's normal meaning—to tell it that I just mean the asterisk literally:

```
$ echo Here is a lone asterisk: \*
Here is a lone asterisk: *
$ echo And here are '*' two "*" more asterisks
And here are * two * more asterisks
```

Shells can run subprocesses to produce text that will then be used as part of their main command, and they can even do math these days. To figure out how many words per line Neal Stephenson fits in the plain-text version of his "In the Beginning Was the Command Line" essay, you can ask the ubiquitous `bash` "Bourne-again" shell, the standard shell on most Linux systems these days, to divide the number of words in the essay by the number of lines and produce a result:

```
$ echo Words/line: $(( $(wc -w <command.txt) / $(wc -l <command.txt) ))
Words/line: 44
```

As is obvious from this example, the rules by which modern shells interpret the special characters in your command line have become quite complex. The manual page for `bash` currently runs to a total of 5,375 lines, or 223 screens full of text in a standard 80×24 terminal window! Obviously, it would lead this chapter far astray if we were to explore even a fraction of the possible ways that a shell can mangle a command that you type.

Instead, to use the command line effectively, you just have to understand two points:

- Special characters are interpreted as special by the shell you are using, like `bash`. They do *not* mean anything special to the operating system itself!
- When passing commands to a shell either locally or—as will be more common in this chapter—across the network, you need to escape the special characters you use so that they are not expanded into unintended values on the remote system.

We will now tackle each of these points in its own section. Keep in mind that we are talking about the common server operating systems here like Linux and OS X, not more primitive systems like Windows, which we will discuss in its own section.

Unix Has No Special Characters

Like many very useful statements, the bold claim of the title of this section is, alas, a lie. There is, in fact, a character that Unix considers special. We will get to that in a moment.

But, in general, Unix has no special characters, and this is a very important fact for you to grasp. If you have used a shell like bash for any great length of time at all, you may have come to view your system as a sort of very powerful and convenient minefield. On the one hand, it makes it very easy to, say, name all of the files in the current directory as arguments to a command; but on the other hand, it can be very difficult to echo a message to the screen that mixes single quotes and double-quotes.

The simple lesson of this section is that the whole set of conventions to which you are accustomed has *nothing* to do with your operating system; they are simply and entirely a behavior of the bash shell, or of whichever of the other popular (or arcane) shells that you are using. It does not matter how familiar the rules seem, or how difficult it is for you to imagine using a Unix-like system without them. If you take bash away, they are simply *not there*.

You can observe this quite simply by taking control of the operating system's process launcher yourself and trying to throw some special characters at a familiar command:

```
>>> import subprocess
>>> args = ['echo', 'Sometimes an', '*', 'just means an', '*']
>>> subprocess.call(args)
Sometimes an * just means an *
```

Here, we are bypassing all of the shell applications that are available for interpreting commands, and we are telling the operating system to start a new process using precisely the list of arguments we have provided. And the process—the echo command, in this case—is getting exactly those characters, instead of having the * turned into a list of file names first.

Though we rarely think about it, the most common "special" character is one we use all the time: the space character! Rather than assume that you actually mean each space character to be passed to the command you are invoking, the shell instead interprets it as the delimiter separating the actual text you want the command to see. This causes endless entertainment when people include spaces in Unix file names, and then try to move the file somewhere else:

```
$ mv Smith Contract.txt ~/Documents
mv: cannot stat `Smith': No such file or directory
mv: cannot stat `Contract.txt': No such file or directory
```

To make the shell understand that you are talking about one file with a space in its name, not two files, you have to contrive something like one of these possible command lines:

```
$ mv Smith\ Contract.txt ~/Documents
$ mv "Smith Contract.txt" ~/Documents
$ mv Smith*Contract.txt ~/Documents
```

That last possibility obviously means something quite different—since it will match any file name that happens to start with Smith and end with Contract.txt, regardless of whether the text between them is a simple space character or some much longer sequence of text—but I have seen many people type it in frustration who are still learning shell conventions and cannot remember how to type a literal space character for the shell.

If you want to convince yourself that none of the characters that the bash shell has taught you to be careful about is special, Listing 16–2 shows a simple shell, written in Python, that treats *only* the space as special but passes everything else through literally to the command.

Listing 16–2. *Shell Supporting Whitespace-Separated Arguments*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - shell.py
# A simple shell, so you can try running commands in the absence of
# any special characters (except for whitespace, used for splitting).

import subprocess

while True:
»   args = raw_input('] ').split()
»   if not args:
»   »   pass
»   elif args == ['exit']:
»   »   break
»   elif args[0] == 'show':
»   »   print "Arguments:", args[1:]
»   else:
»   »   subprocess.call(args)
```

Of course, this means that you cannot use this shell to talk about files with spaces in their names, since without at least one other special character—an escape or quoting character—you cannot make the spaces mean anything but the argument separator! But you can quickly bring up this shell to try out all sorts of special characters of which you have always been afraid, and see that they mean absolutely nothing if passed directly to the common commands you use (the shell in Listing 16–2 uses a] prompt, to make it easy to tell apart from your own shell):

```
$ python shell.py
] echo Hi there!
Hi there!
] echo An asterisk * is not special.
An asterisk * is not special.
] echo The string $HOST is not special, nor are "double quotes".
The string $HOST is not special, nor are "double quotes".
] echo What? No *<>!$ special characters?
What? No *<>!$ special characters?
] show "The 'show' built-in lists its arguments."
Arguments: ['"The', "'show'", 'built-in', 'lists', 'its', 'arguments."']
] exit
```

You can see here absolute evidence that Unix commands—in this case, the `/bin/echo` command that we are calling over and over again—do *not* generally attempt to interpret their arguments as anything other than strings. The echo command happily accepts double-quotes, dollar signs, and asterisks, and treats them all as literal characters. As the foregoing `show` command illustrates, Python is simply reducing our arguments to a list of strings for the operating system to use in creating a new process.

What if we fail to split our command into separate arguments?

```
>>> import subprocess
>>> subprocess.call(['echo hello'])
Traceback (most recent call last):
  ...
OSError: [Errno 2] No such file or directory
```

Do you see what has happened? The operating system does not know that spaces should be special; that is a quirk of shell programs, not of Unix-like operating systems themselves! So the system thinks that it is being asked to run a command literally named `echo` [space] `hello`, and, unless you have created such a file in the current directory, it fails to find it and raises an exception.

Oh—I said at the beginning of this whole section that its whole premise was a lie, and you probably want to know what character is, in fact, special to the system! It turns out that it is the null character—the character having the Unicode and ASCII code zero. This character is used in Unix-like systems to mark the end of each command-line argument in memory. So if you try using a null character in an argument, Unix will think the argument has ended and will ignore the rest of its text. To prevent you from making this mistake, Python stops you in your tracks if you include a null character in a command-line argument:

```
>>> import subprocess
>>> subprocess.call(['echo', 'Sentences can end\0 abruptly.'])
Traceback (most recent call last):
  ...
TypeError: execv() arg 2 must contain only strings
```

Happily, since every command on the system is designed to live within this limitation, you will generally find there is never any reason to put null characters into command-line arguments anyway! (Specifically, they cannot appear in file names for exactly the same reason as they cannot appear in arguments: file names are null-terminated in the operating system implementation.)

Quoting Characters for Protection

In the foregoing section, we used routines in Python's `subprocess` module to directly invoke commands. This was great, and let us pass characters that would have been special to a normal interactive shell. If you have a big list of file names with spaces and other special characters in them, it can be wonderful to simply pass them into a `subprocess` call and have the command on the receiving end understand you perfectly.

But when you are using remote-shell protocols over the network (which, you will recall, is the subject of this chapter!), you are generally going to be talking to a shell like `bash` instead of getting to invoke commands directly like you do through the `subprocess` module. This means that remote-shell protocols will feel more like the `system()` routine from the `os` module, which does invoke a shell to interpret your command line, and therefore involves you in all of the complexities of the Unix command line:

```
>>> import os
>>> os.system('echo *')
Makefile chapter-16.txt formats.ini out.odt source tabify2.py test.py
```

Of course, if the other end of a remote-shell connection is using some sort of shell with which you are unfamiliar, there is little that Python can do. The authors of the Standard Library have no idea how, say, a Motorola DSL router's Telnet-based command line might handle special characters, or even whether it pays attention to quotes at all.

But if the other end of a network connection is a standard Unix shell of the `sh` family, like `bash` or `zsh`, then you are in luck: the fairly obscure Python `pipes` module, which is normally used to build complex shell command lines, contains a helper function that is perfect for escaping arguments. It is called `quote`, and can simply be passed a string:

```
>>> from pipes import quote
>>> print quote("filename")
filename
>>> print quote("file with spaces")
```

```
'file with spaces'
>>> print quote("file 'single quoted' inside!")
"file 'single quoted' inside!"
>>> print quote("danger!; rm -r *")
'danger!; rm -r *'
```

So preparing a command line for remote execution generally just involves running `quote()` on each argument and then pasting the result together with spaces.

Note that using a remote shell with Python does *not* involve you in the terrors of *two* levels of shell quoting! If you have ever tried to build a remote SSH command line that uses fancy quoting, by typing a local command line into your own shell, you will know what I am talking about! The attempt tends to generate a series of experiments like this:

```
$ echo $HOST
guinness
$ ssh asaph echo $HOST
guinness
$ ssh asaph echo \$HOST
asaph
$ ssh asaph echo \\$HOST
guinness
$ ssh asaph echo \\\$HOST
$HOST
$ ssh asaph echo \\\\$HOST
\guinness
```

Every one of these responses is reasonable, as you can demonstrate to yourself if you first use `echo` to see what each command looks like when quoted by the local shell, then paste that text into a remote SSH command line to see how the processed text is handled there. But they can be very tricky to write, and even a practiced Unix shell user can guess wrong when he or she tries to predict what the output should be from the foregoing series of commands!

Fortunately, using a remote-shell protocol through Python does *not* involve two levels of shell like this. Instead, you get to construct a literal string in Python that then directly becomes what is executed by the remote shell; no local shell is involved. (Though, of course, you have to be careful if any string literals in your Python program include backslashes, as usual!)

So if using a shell-within-a-shell has you convinced that passing strings and file names safely to a remote shell is a very hard problem, relax: no local shell will be involved in our following examples.

The Terrible Windows Command Line

Have you read the previous sections on the Unix shell and how arguments are ultimately delivered to a process?

Well, if you are going to be connecting to a Windows machine using a remote-shell protocol, then you can forget everything you have just read. Windows is amazingly primitive: instead of delivering command-line arguments to a new process as separate strings, it simply hands over the text of the entire command line, and makes the process itself try to figure out how the user might have quoted file names with spaces in them!

Of course, merely to survive, people in the Windows world have adopted more or less consistent traditions about how commands will interpret their arguments, so that—for example—you can put double-quotes around a several-word file name and expect nearly all programs to recognize that you are naming one file, not several. Most commands also try to understand that asterisks in a file name are wildcards. But this is *always* a choice made by the program you are running, not by the command prompt.

As we will see, there does exist a very primitive network protocol—the ancient Telnet protocol—that also sends command lines simply as text, like Windows does, so that your program will have to do some kind of escaping if it sends arguments with spaces or special characters in them. But if you are using any sort of modern remote protocol like SSH that lets you send arguments as a list of strings, rather than as a single string, then be aware that on Windows systems all that SSH can do is paste your carefully constructed command line back together and hope that the Windows command can figure it out.

When sending commands to Windows, you might want to take advantage of the `list2cmdline()` routine offered by the Python `subprocess` module. It takes a list of arguments like you would use for a Unix command, and attempts to paste them together—using double-quotes and backslashes when necessary—so that "normal" Windows programs will parse the command line back into exactly the same arguments:

```
>>> from subprocess import list2cmdline
>>> args = ['rename', 'salary "Smith".xls', 'salary-smith.xls']
>>> print list2cmdline(args)
rename "salary \"Smith\".xls" salary-smith.xls
```

Some quick experimentation with your network library and remote-shell protocol of choice (after all, the network library might do Windows quoting for you instead of making you do it yourself) should help you figure out what Windows needs in your situation. For the rest of this chapter, we will make the simplifying assumption that you are connecting to servers that use a modern Unix-like operating system and can keep command-line arguments straight without quoting.

Things Are Different in a Terminal

You will probably talk to more programs than just the shell over your Python-powered remote-shell connection, of course. You will often want to watch the incoming data stream for the information and errors printed out by the commands you are running. And sometimes you will even want to send data back, either to provide the remote programs with input, or to respond to questions and prompts that they present.

When performing tasks like this, you might be surprised to find that programs hang indefinitely without ever finishing the output that you are waiting on, or that data you send seems to not be getting through. To help you through situations like this, a brief discussion of Unix terminals is in order.

A *terminal* typically names a device into which a user types text, and on whose screen the computer's response can be displayed. If a Unix machine has physical serial ports that could possibly host a physical terminal, then the device directory will contain entries like `/dev/ttyS1` with which programs can send and receive strings to that device. But most terminals these days are, in reality, other programs: an `xterm` terminal, or a Gnome or KDE terminal program, or a `PuTTY` client on a Windows machine that has connected via a remote-shell protocol of the kind we will discuss in this chapter.

But the programs running inside the terminal on your laptop or desktop machine still need to know that they are talking to a person—they still need to feel like they are talking through the mechanism of a terminal device connected to a display. So the Unix operating system provides a set of "pseudo-terminal" devices (which might have less confusingly been named "virtual" terminals) with names like `/dev/tty42`. When someone brings up an `xterm` or connects through SSH, the `xterm` or SSH daemon grabs a fresh pseudo-terminal, configures it, and runs the user's shell behind it. The shell examines its standard input, sees that it is a terminal, and presents a prompt since it believes itself to be talking to a person.

■ **Note** Because the noisy teletype machine was the earliest example of a computer terminal, Unix often uses TTY as the abbreviation for a terminal device. That is why the call to test whether your input is a terminal is named `isatty()`!

This is a crucial distinction to understand: the shell presents a prompt because, and only because, it thinks it is connected to a terminal! If you start up a shell and give it a standard input that is not a terminal—like, say, a pipe from another command—then no prompt will be printed, yet it will still respond to commands:

```
$ cat | bash
echo Here we are inside of bash, with no prompt!
Here we are inside of bash, with no prompt!
python
print 'Python has not printed a prompt, either.'
import sys
print 'Is this a terminal?', sys.stdin.isatty()
```

You can see that Python, also, does not print its usual startup banner, nor does it present any prompts.

But then Python also does not seem to be doing *anything* in response to the commands that you are typing. What is going on?

The answer is that since its input is not a terminal, Python thinks that it should just be blindly reading a whole Python script from standard input—after all, its input is a file, and files have whole scripts inside, right? To escape from this endless read from its input that Python is performing, you will have to press Ctrl+D to send an "end-of-file" to `cat`, which will then close its own output—an event that will be seen both by `python` and also by the instance of `bash` that is waiting for Python to complete.

Once you have closed its input, Python will interpret and run the three-line script you have provided (everything past the word `python` in the session just shown), and you will see the results on your terminal, followed by the prompt of the shell that you started at:

```
Python has not printed a prompt, either.
Is this a terminal? False
$
```

There are even changes in how some commands format their output depending on whether they are talking to a terminal. Some commands with long lines of output—the `ps` command comes to mind—will truncate their lines to your terminal width if used interactively, but produce arbitrarily wide output if connected to a pipe or file. And, entertainingly enough, the familiar column-based output of the `ls` command gets turned off and replaced with a file name on each line (which is, you must admit, an easier format for reading by another program) if its output is a pipe or file:

```
$ ls
Makefile        out.odt      test.py
chapter-16.txt  source
formats.ini     tabify2.py
$ ls | cat
Makefile
chapter-16.txt
formats.ini
out.odt
source
```

```
tabify2.py
test.py
```

So what does all of this have to do with network programming?

Well, these two behaviors that we have seen—the fact that programs tend to display prompts if connected to a terminal, but omit them and run silently if they are reading from a file or from the output of another command—also occur at the remote end of the shell protocols that we are considering in this chapter.

A program running behind Telnet, for example, always thinks it is talking to a terminal; so your scripts or programs must always expect to see a prompt each time the shell is ready for input, and so forth. But when you make a connection over the more sophisticated SSH protocol, you will actually have your choice of whether the program thinks that its input is a terminal or just a plain pipe or file. You can test this easily from the command line if there is another computer you can connect to:

```
$ ssh -t asaph
asaph$ echo "Here we are, at a prompt."
Here we are, at a prompt.
asaph$ exit
$ ssh -T asaph
echo "The shell here on asaph sees no terminal; so, no prompt."
The shell here on asaph sees no terminal; so, no prompt.
exit
$
```

So when you spawn a command through a modern protocol like SSH, you need to consider whether you want the program on the remote end thinking that you are a person typing at it through a terminal, or whether it had best think it is talking to raw data coming in through a file or pipe.

Programs are not actually required to act any differently when talking to a terminal; it is just for our convenience that they vary their behavior. They do so by calling the equivalent of the Python `isatty()` call ("is this a teletype?") that you saw in the foregoing example session, and then having "if" statements everywhere that vary their behavior depending on what this call returns. Here are some common ways that they behave differently:

- Programs that are often used interactively will present a human-readable prompt when they are talking to a terminal. But when they think input is coming from a file, they avoid printing a prompt, because otherwise your screen would become littered with hundreds of successive prompts as you ran a long shell script or Python program!

- Sophisticated interactive programs, these days, usually turn on command-line editing when their input is a TTY. This makes many control characters special, because they are used to access the command-line history and perform editing commands. When they are not under the control of a terminal, these same programs turn command-line editing off and absorb control characters as normal parts of their input stream.

- Many programs read only one line of input at a time when listening to a terminal, because humans like to get an immediate response to every command they type. But when reading from a pipe or file, these same programs will wait until thousands of characters have arrived before they try to interpret their first batch of input. As we just saw, `bash` stays in line-at-a-time mode even if its input is a file, but Python decided it wanted to read a whole Python script from its input before trying to execute even its first line.

- It is even more common for programs to adjust their output based on whether they are talking to a terminal. If a user might be watching, they want each line, or even each character, of output to appear immediately. But if they are talking to a mere file or pipe, they will wait and batch up large chunks of output and more efficiently send the whole chunk at one time.

Both of the last two issues, which involve buffering, cause all sorts of problems when you take a process that you usually do manually and try to automate it—because in doing so you often move from terminal input to input provided through a file or pipe, and suddenly you find that the programs behave quite differently, and might even seem to be hanging because "print" statements are not producing immediate output, but are instead saving up their results to push out all at once when their output buffer is full.

You can see this easily with a simple Python program (since Python is one of the applications that decides whether to buffer its output based on whether it is talking to a terminal) that prints a message, waits for a line of input, and then prints again:

```
$ python -c 'print "talk:"; s = raw_input(); print "you said", s'
talk:
hi
you said hi
$ python -c 'print "talk:"; s = raw_input(); print "you said", s' | cat
hi
talk:
you said hi
```

You can see that in the first instance, when Python knew its output was a terminal, it printed `talk:` immediately. But in the second instance, its output was a pipe to the `cat` command, and so it decided that it could save up the results of that first print statement and batch them together with the rest of the program's output, so that both lines of output appeared only once you had provided your input and the program was ending.

The foregoing problem is why many carefully written programs, both in Python and in other languages, frequently call `flush()` on their output to make sure that anything waiting in a buffer goes ahead and gets sent out, regardless of whether the output looks like a terminal.

So those are the basic problems with terminals and buffering: programs change their behavior, often in idiosyncratic ways, when talking to a terminal (think again of the `ls` example), and they often start heavily buffering their output if they think they are writing to a file or pipe.

Terminals Do Buffering

Beyond the program-specific behaviors just described, there are additional problems raised by terminals.

For example, what happens when you want a program to be reading your input one character at a time, but the Unix terminal device itself is buffering your keystrokes to deliver them as a whole line? This common problem happens because the Unix terminal defaults to "canonical" input processing, where it lets the user enter a whole line, and even edit it by backspacing and re-typing, before finally pressing "Enter" and letting the program see what he or she has typed.

If you want to turn off canonical processing so that a program can see every individual character as it is typed, you can use the `stty` "Set TTY settings" command to disable it:

```
$ stty -icanon
```

Another problem is that Unix terminals traditionally supported a pair of keystrokes for pausing the output stream so that the user could read something on the screen before it scrolled off and was

replaced by more text. Often these were the characters Ctrl+S for "Stop" and Ctrl+Q for "Keep going," and it was a source of great annoyance that if binary data worked its way into an automated Telnet connection that the first Ctrl+S that happened to pass across the channel would pause the terminal and probably ruin the session.

Again, this setting can be turned off with `stty`:

```
$ stty -ixon -ixoff
```

Those are the two biggest problems you will run into with terminals doing buffering, but there are plenty of less famous settings that can also cause you grief. Because there are so many—and because they vary between Unix implementations—the `stty` command actually supports two modes, `cooked` and `raw`, that turn dozens of settings like `icanon` and `ixon` on and off together:

```
$ stty raw
$ stty cooked
```

In case you make your terminal settings a hopeless mess after some experimentation, most Unix systems provide a command for resetting the terminal back to reasonable, sane settings (and note that if you have played with `stty` too severely, you might need to hit Ctrl+J to submit the `reset` command, since your Return key, whose equivalent is Ctrl+M, actually only functions to submit commands because of a terminal setting called `icrnl`!):

```
$ reset
```

If, instead of trying to get the terminal to behave across a Telnet or SSH session, you happen to be talking to a terminal from Python, check out the `termios` module that comes with the Standard Library. By puzzling through its example code and remembering how Boolean bitwise math works, you should be able to control all of the same settings that we just accessed through the `stty` command.

This book lacks the space to look at terminals in any more detail (since one or two chapters of examples could easily be inserted right here to cover all of the interesting techniques and cases), but there are lots of great resources for learning more about them—a classic is Chapter 19, "Pseudo Terminals," of W. Richard Stevens' *Advanced Programming in the UNIX Environment*.

Telnet

This brief section is all you will find in this book about the ancient Telnet protocol. Why? Because it is insecure: anyone watching your Telnet packets fly by will see your username, password, and everything you do on the remote system. It is clunky. And it has been completely abandoned for most systems administration.

THE TELNET PROTOCOL

Purpose: **Remote shell access**

Standard: **RFC 854 (1989)**

Runs atop: **TCP/IP**

Default port: **23**

Library: `telnetlib`

Exceptions: `socket.error`, `socket.gaierror`

The only time I ever find myself needing Telnet is when speaking to small embedded systems, like a Linksys router or DSL modem or network switch. In case you are having to write a Python program that has to speak Telnet to one of these devices, here are a few pointers on using the Python `telnetlib`.

First, you have to realize that all Telnet does is to establish a channel—in fact, a fairly plain TCP socket (see Chapter 3)—and to send the things you type, and receive the things the remote system says, back and forth across that channel. This means that Telnet is ignorant of all sorts of things of which you might expect a remote-shell protocol to be aware.

For example, it is conventional that when you Telnet to a Unix machine, you are presented with aa `login:` prompt at which you type your username, and a `password:` prompt where you enter your password. The small embedded devices that still use Telnet these days might follow a slightly simpler script, but they, too, often ask for some sort of password or authentication. But the point is that Telnet knows nothing about this! To your Telnet client, `password:` is just nine random characters that come flying across the TCP connection and that it must print to your screen. It has no idea that you are being prompted, that you are responding, or that in a moment the remote system will know who you are.

The fact that Telnet is ignorant about authentication has an important consequence: you cannot type anything on the command line itself to get yourself pre-authenticated to the remote system, nor avoid the login and password prompts that will pop up when you first connect! If you are going to use plain Telnet, you are going to have to somehow watch the incoming text for those two prompts (or however many the remote system supplies) and issue the correct replies.

Obviously, if systems vary in what username and password prompts they present, then you can hardly expect standardization in the error messages or responses that get sent back when your password fails. That is why Telnet is so hard to script and program from a language like Python and a library like `telnetlib`. Unless you know every single error message that the remote system could produce to your login and password—which might not just be its "bad password" message, but also things like "cannot spawn shell: out of memory," "home directory not mounted," and "quota exceeded: confining you to a restricted shell"—your script will sometimes run into situations where it is waiting to see either a command prompt or else an error message it recognizes, and will instead simply wait forever without seeing anything on the inbound character stream that it recognizes.

So if you are using Telnet, then you are playing a text game: you watch for text to arrive, and then try to reply with something intelligible to the remote system. To help you with this, the Python `telnetlib` provides not only basic methods for sending and receiving data, but also a few routines that will watch and wait for a particular string to arrive from the remote system. In this respect, `telnetlib` is a little bit like the third-party Python `pexpect` library that we mentioned early in this chapter, and therefore a bit like the venerable Unix `expect` command that largely exists because Telnet makes us play a textual pattern-matching game. In fact, one of these `telnetlib` routines is, in honor of its predecessor, named `expect()`!

Listing 16–3 connects to localhost, which in this case is my Ubuntu laptop, where I have just run aptitude install telnetd so that a Telnet daemon is now listening on its standard port 23. Yes, I actually changed my password to mypass to test the scripts in this chapter; and, yes, I un-installed telnetd and changed my password again immediately after!

Listing 16–3. *Logging In to a Remote Host Using Telnet*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - telnet_login.py
# Connect to localhost, watch for a login prompt, and try logging in

import telnetlib

t = telnetlib.Telnet('localhost')
# t.set_debuglevel(1)        # uncomment this for debugging messages

t.read_until('login:')
t.write('brandon\n')
t.read_until('assword:')     # let "P" be capitalized or not
t.write('mypass\n')
n, match, previous_text = t.expect([r'Login incorrect', r'\$'], 10)
if n == 0:
»   print "Username and password failed - giving up"
else:
»   t.write('exec uptime\n')
»   print t.read_all()       # keep reading until the connection closes
```

If the script is successful, it shows you what the simple uptime command prints on the remote system:

```
$ python telnet_login.py
10:24:43 up 5 days, 12:13, 14 users,  load average: 1.44, 0.91, 0.73
```

The listing shows you the general structure of a session powered by telnetlib. First, a connection is established, which is represented in Python by an instance of the Telnet object. Here only the hostname is specified, though you can also provide a port number to connect to some other service port than standard Telnet.

You can call set_debuglevel(1) if you want your Telnet object to print out all of the strings that it sends and receives during the session. This actually turned out to be important for writing even the very simple script shown in the listing, because in two different cases it got hung up, and I had to re-run it with debugging messages turned on so that I could see the actual output and fix the script. (Once I was failing to match the exact text that was coming back, and once I forgot the '\r' at the end of the uptime command.) I generally turn off debugging only once a program is working perfectly, and turn it back on whenever I want to do more work on the script.

Note that Telnet does not disguise the fact that its service is backed by a TCP socket, and will pass through to your program any socket.error and socket.gaierror exceptions that are raised.

Once the Telnet session is established, interaction generally falls into a receive-and-send pattern, where you wait for a prompt or response from the remote end, then send your next piece of information. The listing illustrates two methods of waiting for text to arrive:

- The very simple read_until() method watches for a literal string to arrive, then returns a string providing all of the text that it received from the moment it started listing until the moment it finally saw the string you were waiting for.

- The more powerful and sophisticated expect() method takes a list of Python regular expressions. Once the text arriving from the remote end finally adds up to something that matches one of the regular expressions, expect() returns three items: the index in your list of the pattern that matched, the regular expression SRE_Match object itself, and the text that was received leading up to the matching text. For more information on what you can do with a SRE_Match, including finding the values of any sub-expressions in your pattern, read the Standard Library documentation for the re module.

Regular expressions, as always, have to be written carefully. When I first wrote this script, I used '$' as the expect() pattern that watched for the shell prompt to appear—which, of course, is a special character in a regular expression! So the corrected script shown in the listing escapes the $ so that expect() actually waits until it sees a dollar sign arrive from the remote end.

If the script sees an error message because of an incorrect password—and does not get stuck waiting forever for a login or password prompt that never arrives or that looks different than it was expecting—then it exits:

```
$ python telnet_login.py
Username and password failed - giving up
```

If you wind up writing a Python script that has to use Telnet, it will simply be a larger or more complicated version of the same simple pattern shown here.

Both read_until() and expect() take an optional second argument named timeout that places a maximum limit on how long the call will watch for the text pattern before giving up and returning control to your Python script. If they quit and give up because of the timeout, they do not raise an error; instead—awkwardly enough—they just return the text they have seen so far, and leave it to you to figure out whether that text contains the pattern!

There are a few odds and ends in the Telnet object that we need not cover here. You will find them in the telnetlib Standard Library documentation—including an interact() method that lets the user "talk" directly over your Telnet connection using the terminal! This kind of call was very popular back in the old days, when you wanted to automate login but then take control and issue normal commands yourself.

The Telnet protocol does have a convention for embedding control information, and telnetlib follows these protocol rules carefully to keep your data separate from any control codes that appear. So you can use a Telnet object to send and receive all of the binary data you want, and ignore the fact that control codes might be arriving as well. But if you are doing a sophisticated Telnet-based project, then you might need to process options.

Normally, each time a Telnet server sends an option request, telnetlib flatly refuses to send or receive that option. But you can provide a Telnet object with your own callback function for processing options; a modest example is shown in Listing 16–4. For most options, it simply re-implements the default telnetlib behavior and refuses to handle any options (and always remember to respond to each option one way or another; failing to do so will often hang the Telnet session as the server waits forever for your reply). But if the server expresses interest in the "terminal type" option, then this client sends back a reply of "mypython," which the shell command it runs after logging in then sees as its $TERM environment variable.

Listing 16–4. *How to Process Telnet Option Codes*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - telnet_codes.py
# How your code might look if you intercept Telnet options yourself

from telnetlib import Telnet, IAC, DO, DONT, WILL, WONT, SB, SE, TTYPE
```

```
def process_option(tsocket, command, option):
»   if command == DO and option == TTYPE:
»   »   tsocket.sendall(IAC + WILL + TTYPE)
»   »   print 'Sending terminal type "mypython"'
»   »   tsocket.sendall(IAC + SB + TTYPE + '\0' + 'mypython' + IAC + SE)
»   elif command in (DO, DONT):
»   »   print 'Will not', ord(option)
»   »   tsocket.sendall(IAC + WONT + option)
»   elif command in (WILL, WONT):
»   »   print 'Do not', ord(option)
»   »   tsocket.sendall(IAC + DONT + option)

t = Telnet('localhost')
# t.set_debuglevel(1)        # uncomment this for debugging messages

t.set_option_negotiation_callback(process_option)
t.read_until('login:', 5)
t.write('brandon\n')
t.read_until('assword:', 5)  # so P can be capitalized or not
t.write('mypass\n')
n, match, previous_text = t.expect([r'Login incorrect', r'\$'], 10)
if n == 0:
»   print "Username and password failed - giving up"
else:
»   t.write('exec echo $TERM\n')
»   print t.read_all()
```

For more details about how Telnet options work, again, you can consult the relevant RFCs.

SSH: The Secure Shell

The SSH protocol is one of the best-known examples of a secure, encrypted protocol among modern system administrators (HTTPS is probably the very best known).

THE SSH PROTOCOL

Purpose: **Secure remote shell, file transfer, port forwarding**

Standard: **RFC 4250–4256 (2006)**

Runs atop: **TCP/IP**

Default port: **22**

Library: `paramiko`

Exceptions: `socket.error`, `socket.gaierror`, `paramiko.SSHException`

SSH is descended from an earlier protocol that supported "remote login," "remote shell," and "remote file copy" commands named `rlogin`, `rsh`, and `rcp`, which in their time tended to become much more popular than Telnet at sites that supported them. You cannot imagine what a revelation `rcp` was, in

particular, unless you have spent hours trying to transfer a file between computers armed with only Telnet and a script that tries to type your password for you, only to discover that your file contains a byte that looks like a control character to Telnet or the remote terminal, and have the whole thing hang until you add a layer of escaping (or figure out how to disable both the Telnet escape key and all interpretation taking place on the remote terminal).

But the best feature of the `rlogin` family was that they did not just echo username and password prompts without actually knowing the meaning of what was going on. Instead, they stayed involved through the process of authentication, and you could even create a file in your home directory that told them "when someone named `brandon` tries to connect from the `asaph` machine, just let them in without a password." Suddenly, system administrators and Unix users alike received back hours of each month that would otherwise have been spent typing their password. Suddenly, you could copy ten files from one machine to another nearly as easily as you could have copied them into a local folder.

SSH has preserved all of these great features of the early remote-shell protocol, while bringing bulletproof security and hard encryption that is trusted worldwide for administering critical servers. This chapter will focus on SSH-2, the most recent version of the protocol, and on the `paramiko` Python package that can speak the protocol—and does it so successfully that it has actually been ported to Java, too, because people in the Java world wanted to be able to use SSH as easily as we do when using Python.

An Overview of SSH

You have reached a point in this book where something very interesting happens: we encounter a new layer of multiplexing.

The first section of this book talked a lot about multiplexing—about how UDP (Chapter 2) and TCP (Chapter 3) take the underlying IP protocol, which has no concept that there might actually be several users or applications on a single computer that need to communicate, and add the concept of UDP and TCP port numbers, so that several different conversations between a pair of IP addresses can take place at the same time.

Once that basic level of multiplexing was established, we more or less left the topic behind. Through more than a dozen chapters now, we have studied protocols that take a UDP or TCP connection and then happily use it for exactly one thing—downloading a web page, or transmitting an e-mail, but never trying to do several things at the same time over a single socket.

But as we now arrive at SSH, we reach a protocol so sophisticated that it actually implements its own rules for multiplexing, so that several "channels" of information can all share the same SSH socket. Every block of information SSH sends across its socket is labeled with a "channel" identifier so that several conversations can share the socket.

There are at least two reasons sub-channels make sense. First, even though the channel ID takes up a bit of bandwidth for every single block of information transmitted, the additional data is small compared to how much extra information SSH has to transmit to negotiate and maintain encryption anyway. Second, channels make sense because the *real* expense of an SSH connection is setting it up. Host key negotiation and authentication can together take up several seconds of real time, and once the connection is established, you want to be able to use it for as many operations as possible. Thanks to the SSH notion of a channel, you can amortize the high cost of connecting by performing many operations before you let the connection close.

Once connected, you can create several kinds of channels:

- An interactive shell session, like that supported by Telnet
- The individual execution of a single command
- A file-transfer session letting you browse the remote filesystem
- A port-forward that intercepts TCP connections

We will learn about all of these kinds of channels in the following sections.

SSH Host Keys

When an SSH client first connects to a remote host, they exchange temporary public keys that let them encrypt the rest of their conversation without revealing any information to any watching third parties. Then, before the client is willing to divulge any further information, it demands proof of the remote server's identity. This makes good sense as a first step: if you are really talking to a hacker who has temporarily managed to grab the remote server's IP, you do not want SSH to divulge even your username—much less your password!

As we saw in Chapter 6, one answer to the problem of machine identity on the Internet is to build a public-key infrastructure. First you designate a set of organizations called "certificate authorities" that can issue certs; then you install a list of their public keys in all of the web browsers and other SSL clients in existence; then those organizations charge you money to verify that you really are `google.com` and that you deserve to have your `google.com` SSL certificate signed; and then, finally, you can install the certificate on your web server, and everyone will trust that you are really `google.com`.

There are many problems with this system from the point of view of SSH. While it is true that you can build a public-key infrastructure internal to an organization, where you distribute your own signing authority's certificates to your web browsers or other applications and then can sign your own server certificates without paying a third party, a public-key infrastructure is still considered too cumbersome a process for something like SSH; server administrators want to set up, use, and tear down servers all the time, without having to talk to a central authority first.

So SSH has the idea that each server, when installed, creates its own random public-private key pair that is not signed by anybody. Instead, one of two approaches is taken to key distribution:

- A system administrator writes a script that gathers up all of the host public keys in an organization, creates an `ssh_known_hosts` listing them all, and places this file in the `/etc/sshd` directory on every system in the organization. They might also make it available to any desktop clients, like the PuTTY command under Windows. Now every SSH client will know about every SSH host key before they even connect for the first time.
- Abandon the idea of knowing host keys ahead of time, and instead memorize them at the moment of first connection. Users of the SSH command line will be very familiar with this: the client says it does not recognize the host to which you are connecting, you reflexively answer "yes," and its key gets stored in your `~/.ssh/known_hosts` file. You actually have no guarantee that you are really talking to the host you think it is; but at least you will be guaranteed that every subsequent connection you ever make to that machine is going to the right place, and not to other servers that someone is swapping into place at the same IP address. (Unless, of course, they have stolen your host keys!)

The familiar prompt from the SSH command line when it sees an unfamiliar host looks like this:

```
$ ssh asaph.rhodesmill.org
The authenticity of host 'asaph.rhodesmill.org (74.207.234.78)'
   can't be established.
RSA key fingerprint is 85:8f:32:4e:ac:1f:e9:bc:35:58:c1:d4:25:e3:c7:8c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'asaph.rhodesmill.org,74.207.234.78' (RSA)
   to the list of known hosts.
```

That "yes" answer buried deep on the next-to-last full line is the answer that I typed giving SSH the go-ahead to make the connection and remember the key for next time. If SSH ever connects to a host and sees a different key, its reaction is quite severe:

```
$ ssh asaph.rhodesmill.org
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
```

This message will be familiar to anyone who has ever had to re-build a server from scratch, and forgets to save the old SSH keys and lets new ones be generated by the re-install. It can be painful to go around to all of your SSH clients and remove the offending old key so that they will quietly learn the new one upon reconnection.

The `paramiko` library has full support for all of the normal SSH tactics surrounding host keys. But its default behavior is rather spare: it loads no host-key files by default, and will then, of course, raise an exception for the very first host to which you connect because it will not be able to verify its key! The exception that it raises is a bit un-informative; it is only by looking at the fact that it comes from inside the `missing_host_key()` function that I usually recognize what has caused the error:

```
>>> import paramiko
>>> client = paramiko.SSHClient()
>>> client.connect('my.example.com', username='test')
Traceback (most recent call last):
  ...
  File ".../paramiko/client.py", line 85, in missing_host_key
»   raise SSHException('Unknown server %s' % hostname)
paramiko.SSHException: Unknown server my.example.com
```

To behave like the normal SSH command, load both the system and the current user's known-host keys before making the connection:

```
>>> client.load_system_host_keys()
>>> client.load_host_keys('/home/brandon/.ssh/known_hosts')
>>> client.connect('my.example.com', username='test')
```

The `paramiko` library also lets you choose how you handle unknown hosts. Once you have a client object created, you can provide it with a decision-making class that is asked what to do if a host key is not recognized. You can build these classes yourself by inheriting from the `MissingHostKeyPolicy` class:

```
>>> class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
...     def missing_host_key(self, client, hostname, key):
...         return
...
>>> client.set_missing_host_key_policy(AllowAnythingPolicy())
>>> client.connect('my.example.com', username='test')
```

Note that, through the arguments to the `missing_host_key()` method, you receive several pieces of information on which to base your decision; you could, for example, allow connections to machines on your own server subnet without a host key, but disallow all others.

Inside `paramiko` there are also several decision-making classes that already implement several basic host-key options:

- `paramiko.AutoAddPolicy`: Host keys are automatically added to your user host-key store (the file `~/.ssh/known_hosts` on Unix systems) when first encountered, but any change in the host key from then on will raise a fatal exception.

- `paramiko.RejectPolicy`: Connecting to hosts with unknown keys simply raises an exception.
- `paramiko.WarningPolicy`: An unknown host causes a warning to be logged, but the connection is then allowed to proceed.

When writing a script that will be doing SSH, I always start by connecting to the remote host "by hand" with the normal `ssh` command-line tool so that I can answer "yes" to its prompt and get the remote host's key in my host-keys file. That way, my programs should never have to worry about handling the case of a missing key, and can die with an error if they encounter one.

But if you like doing things less by-hand than I do, then the `AutoAddPolicy` might be your best bet: it never needs human interaction, but will at least assure you on subsequent encounters that you are still talking to the same machine as before. So even if the machine is a Trojan horse that is logging all of your interactions with it and secretly recording your password (if you are using one), it at least must prove to you that it holds the same secret key every time you connect.

SSH Authentication

The whole subject of SSH authentication is the topic of a large amount of good documentation, as well as articles and blog posts, all available on the Web. Information abounds about configuring common SSH clients, setting up an SSH server on a Unix or Windows host, and using public keys to authenticate yourself so that you do not have to keep typing your password all the time. Since this chapter is primarily about how to "speak SSH" from Python, I will just briefly outline how authentication works.

There are generally three ways to prove your identity to a remote server you are contacting through SSH:

- You can provide a username and password.
- You can provide a username, and then have your client successfully perform a public-key challenge-response. This clever operation manages to prove that you are in possession of a secret "identity" key without actually exposing its contents to the remote system.
- You can perform Kerberos authentication. If the remote system is set up to allow Kerberos (which actually seems extremely rare these days), and if you have run the `kinit` command-line tool to prove your identity to one of the master Kerberos servers in the SSH server's authentication domain, then you should be allowed in without a password.

Since option 3 is very rare, we will concentrate on the first two.

Using a username and password with `paramiko` is very easy—you simply provide them in your call to the `connect()` method:

```
>>> client.connect('my.example.com', username='brandon', password=mypass)
```

Public-key authentication, where you use `ssh-keygen` to create an "identity" key pair (which is typically stored in your `~/.ssh` directory) that can be used to authenticate you without a password, makes the Python code even easier!

```
>>> client.connect('my.example.com')
```

If your identity key file is stored somewhere other than in the normal `~/.ssh/id_rsa` file, then you can provide its file name—or a whole Python list of file names—to the `connect()` method manually:

```
>>> client.connect('my.example.com',
```

```
...     key_filename='/home/brandon/.ssh/id_sysadmin')
```

Of course, per the normal rules of SSH, providing a public-key identity like this will work only if you have appended the public key in the `id_sysadmin.pub` file to your “authorized hosts” file on the remote end, typically named something like this:

```
/home/brandon/.ssh/authorized_keys
```

If you have trouble getting public-key authentication to work, always check the file permissions on both your remote `.ssh` directory and also the files inside; some versions of the SSH server will get upset if they see that these files are group-readable or group-writable. Using mode 0700 for the `.ssh` directory and 0600 for the files inside will often make SSH happiest. The task of copying SSH keys to other accounts has actually been automated in recent versions, through a small command that will make sure that the file permissions get set correctly for you:

```
ssh-copy-id -i ~/.ssh/id_rsa.pub myaccount@example.com
```

Once the `connect()` method has succeeded, you are now ready to start performing remote operations, all of which will be forwarded over the same physical socket without requiring re-negotiation of the host key, your identity, or the encryption that protects the SSH socket itself!

Shell Sessions and Individual Commands

Once you have a connected SSH client, the entire world of SSH operations is open to you. Simply by asking, you can access remote-shell sessions, run individual commands, commence file-transfer sessions, and set up port forwarding. We will look at each of these operations in turn.

First, SSH can set up a raw shell session for you, running on the remote end inside a pseudo-terminal so that programs act like they normally do when they are interacting with the user at a terminal. This kind of connection behaves very much like a Telnet connection; take a look at Listing 16–5 for an example, which pushes a simple `echo` command at the remote shell, and then asks it to exit.

Listing 16–5. *Running an Interactive Shell Under SSH*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - ssh_simple.py
# Using SSH like Telnet: connecting and running two commands

import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return
client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test')  # password='')

channel = client.invoke_shell()
stdin = channel.makefile('wb')
stdout = channel.makefile('rb')

stdin.write('echo Hello, world\rexit\r')
print stdout.read()

client.close()
```

You will see that this awkward session bears all of the scars of a program operating over a terminal. Instead of being able to neatly encapsulate each command and separate its arguments in Python, it has to use spaces and carriage returns and trust the remote shell to divide things back up properly.

■ **Note** All of the commands in this section simply connect to the `localhost` IP address, 127.0.0.1, and thus should work fine if you are on a Linux or Mac with an SSH server installed, and you have copied your SSH identity public key into your authorized-keys file. If, instead, you want to use these scripts to connect to a remote SSH server, simply change the host given in the `connect()` call.

Also, if you actually run this command, you will see that the commands you type are actually echoed to you *twice*, and that there is no obvious way to separate these command echoes from the actual command output:

```
Ubuntu 10.04.1 LTS
Last login: Mon Sep  6 01:10:36 2010 from 127.0.0.9
echo Hello, world
exit
test@guinness:~$ echo Hello, world
Hello, world
test@guinness:~$ exit
logout
```

Do you see what has happened? Because we did not wait for a shell prompt before issuing our `echo` and `exit` commands (which would have required a loop doing repeated `read()` calls), our command text made it to the remote host while it was still in the middle of issuing its welcome messages. Because the Unix terminal is by default in a “cooked” state, where it echoes the user's keystrokes, the commands got printed back to us, just beneath the “Last login” line.

Then the actual `bash` shell started up, set the terminal to “raw” mode because it likes to offer its own command-line editing interface, and then started reading your commands character by character. And, because it assumes that you want to see what you are typing (even though you are actually finished typing and it is just reading the characters from a buffer that is several milliseconds old), it echoes each command back to the screen a *second* time.

And, of course, without a good bit of parsing and intelligence, we would have a hard time writing a Python routine that could pick out the actual command output—the words `Hello, world`—from the rest of the output we are receiving back over the SSH connection.

Because of all of these quirky, terminal-dependent behaviors, you should generally avoid ever using `invoke_shell()` unless you are actually writing an interactive terminal program where you let a live user type commands.

A much better option for running remote commands is to use `exec_command()`, which, instead of starting up a whole shell session, just runs a single command, giving you control of its standard input, output, and error streams just as though you had run it using the `subprocess` module in the Standard Library. A script demonstrating its use is shown in Listing 16–6. The difference between `exec_command()` and a local subprocess (besides, of course, the fact that the command runs over on the remote machine!) is that you do *not* get the chance to pass command-line arguments as separate strings; instead, you have to pass a whole command line for interpretation by the shell on the remote end.

Listing 16–6. *Running Individual SSH Commands*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - ssh_commands.py
# Running separate commands instead of using a shell

import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
»   def missing_host_key(self, client, hostname, key):
»   »   return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test')  # password='')

for command in 'echo "Hello, world!"', 'uname', 'uptime':
»   stdin, stdout, stderr = client.exec_command(command)
»   stdin.close()
»   print repr(stdout.read())
»   stdout.close()
»   stderr.close()

client.close()
```

As was just mentioned, you might find the `quotes()` function from the Python `pipes` module to be helpful if you need to quote command-line arguments so that spaces containing file names and special characters are interpreted correctly by the remote shell.

Every time you start a new SSH shell session with `invoke_shell()`, and every time you kick off a command with `exec_command()`, a new SSH "channel" is created behind the scenes, which is what provides the file-like Python objects that let you talk to the remote command's standard input, output, and error. Channels, as just explained, can run in parallel, and SSH will cleverly interleave their data on your single SSH connection so that all of the conversations happen simultaneously without ever becoming confused.

Take a look at Listing 16–7 for a very simple example of what is possible. Here, two "commands" are kicked off remotely, which are each a simple shell script with some `echo` commands interspersed with pauses created by calls to `sleep`. If you want, you can pretend that these are really filesystem commands that return data as they walk the filesystem, or that they are CPU-intensive operations that only slowly generate and return their results. The difference does not matter at all to SSH: what matters is that the channels are sitting idle for several seconds at a time, then coming alive again as more data becomes available.

Listing 16–7. *SSH Channels Run in Parallel*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - ssh_threads.py
# Running two remote commands simultaneously in different channels

import threading
import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
»   def missing_host_key(self, client, hostname, key):
```

```
        return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test')  # password='')

def read_until_EOF(fileobj):
    s = fileobj.readline()
    while s:
        print s.strip()
        s = fileobj.readline()

out1 = client.exec_command('echo One;sleep 2;echo Two;sleep 1;echo Three')[1]
out2 = client.exec_command('echo A;sleep 1;echo B;sleep 2;echo C')[1]
thread1 = threading.Thread(target=read_until_EOF, args=(out1,))
thread2 = threading.Thread(target=read_until_EOF, args=(out2,))
thread1.start()
thread2.start()
thread1.join()
thread2.join()

client.close()
```

In order to be able to process these two streams of data simultaneously, we are kicking off two threads, and are handing each of them one of the channels from which to read. They each print out each line of new information as soon as it arrives, and finally exit when the `readline()` command indicates end-of-file by returning an empty string. When run, this script should return something like this:

```
$ python ssh_threads.py
One
A
B
Two
Three
C
```

So there you have it: SSH channels over the same TCP connection are completely independent, can each receive (and send) data at their own pace, and can close independently when the particular command that they are talking to finally terminates.

The same is true of the features we are about to look at—file transfer and port forwarding—so keep in mind as you read our last two examples that all of these kinds of communications can happen simultaneously without your having to open more than one SSH connection to hold all of the channels of data.

SFTP: File Transfer Over SSH

Version 2 of the SSH protocol includes a sub-protocol called the "SSH File Transfer Protocol" (SFTP) that lets you walk the remote directory tree, create and delete directories and files, and copy files back and forth from the local to the remote machine. The capabilities of SFTP are so complex and complete, in fact, that they support not only simple file-copy operations, but can power graphical file browsers and can even let the remote filesystem be mounted locally! (Google for the `sshfs` system for details.)

The SFTP protocol is an incredible boon to those of us who once had to copy files using brittle scripts that tried to send data across Telnet through very careful escaping of binary data! And instead of

making you power up its own `sftp` command line each time you want to move files, SSH follows the tradition of RSH by providing an `scp` command-line tool that acts just like the traditional `cp` command but lets you prefix any file name with `hostname:` to indicate that it exists on the remote machine. This means that remote copy commands stay in your command-line history just like your other shell commands, rather than being lost to the separate history buffer of a separate command prompt that you have to invoke and then quit out of (which was a great annoyance of traditional FTP clients).

And, of course, the great and crowning achievement of SFTP and the `sftp` and `scp` commands is that they not only support password authentication, but also let you copy files using exactly the same public-key mechanism that lets you avoid typing your password over and over again when running remote commands with the `ssh` command!

If you look briefly over Chapter 17 on the old FTP system, you will get a good idea of the sorts of operations that SFTP supports. In fact, most of the SFTP commands have the same names as the local commands that you already run to manipulate files on your Unix shell account, like `chmod` and `mkdir`, or have the same names as Unix system calls that you might be familiar with through the Python `os` module, like `lstat` and `unlink`. Because these operations are so familiar, I never need any other support in writing SFTP commands than is provided by the bare `paramiko` documentation for the Python SFTP client: http:`//www.lag.net/paramiko/docs/paramiko.SFTPClient-class`.

Here are the main things to remember when doing SFTP:

- The SFTP protocol is stateful, just like FTP, and just like your normal shell account. So you can either pass all file and directory names as absolute paths that start at the root of the filesystem, or use `getcwd()` and `chdir()` to move around the filesystem and then use paths that are relative to the directory in which you have arrived.
- You can open a file using either the `file()` or `open()` method (just like Python has a built-in function that lives under both names), and you get back a file-like object connected to an SSH channel that runs independently of your SFTP channel. That is, you can keep issuing SFTP commands, you can move around the filesystem and copy or open further files, and the original channel will still be connected to its file and ready for reading or writing.
- Because each open remote file gets an independent channel, file transfers can happen asynchronously; you can open many remote files at once and have them all streaming down to your disk drive, or open new files and be sending data the other way. Be careful that you recognize this, or you might open so many channels at once that each one slows to a crawl.
- Finally, keep in mind that no shell expansion is done on any of the file names you pass across SFTP. If you try using a file name like `*` or one that has spaces or special characters, they are simply interpreted as part of the file name. No shell is involved when using SFTP; you are getting to talk right to the remote filesystem thanks to the support inside the SSH server itself. This means that any support for pattern-matching that you want to provide to the user has to be through fetching the directory contents yourself and then checking their pattern against each one, using a routine like those provided in `fnmatch` in the Python Standard Library.

A very modest example SFTP session is shown in Listing 16–8. It does something simple that system administrators might often need (but, of course, that they could just as easily accomplish with an `scp` command): it connects to the remote system and copies `messages` log files out of the `/var/log` directory, perhaps for scanning or analysis on the local machine.

Listing 16–8. *Listing a Directory and Fetching Files with SFTP*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 16 - sftp.py
# Fetching files with SFTP

import functools
import paramiko

class AllowAnythingPolicy(paramiko.MissingHostKeyPolicy):
    def missing_host_key(self, client, hostname, key):
        return

client = paramiko.SSHClient()
client.set_missing_host_key_policy(AllowAnythingPolicy())
client.connect('127.0.0.1', username='test')  # password='')

def my_callback(filename, bytes_so_far, bytes_total):
    print 'Transfer of %r is at %d/%d bytes (%.1f%%)' % (
        filename, bytes_so_far, bytes_total, 100. * bytes_so_far / bytes_total)

sftp = client.open_sftp()
sftp.chdir('/var/log')
for filename in sorted(sftp.listdir()):
    if filename.startswith('messages.'):
        callback_for_filename = functools.partial(my_callback, filename)
        sftp.get(filename, filename, callback=callback_for_filename)

client.close()
```

Note that, although I made a big deal of talking about how each file that you open with SFTP uses its own independent channel, the simple `get()` and `put()` convenience functions provided by `paramiko`—which are really lightweight wrappers for an `open()` followed by a loop that reads and writes—do *not* attempt any asynchrony, but instead just block and wait until each whole file has arrived. This means that the foregoing script calmly transfers one file at a time, producing output that looks something like this:

```
$ python sftp.py
Transfer of 'messages.1' is at 32768/128609 bytes (25.5%)
Transfer of 'messages.1' is at 65536/128609 bytes (51.0%)
Transfer of 'messages.1' is at 98304/128609 bytes (76.4%)
Transfer of 'messages.1' is at 128609/128609 bytes (100.0%)
Transfer of 'messages.2.gz' is at 32768/40225 bytes (81.5%)
Transfer of 'messages.2.gz' is at 40225/40225 bytes (100.0%)
Transfer of 'messages.3.gz' is at 28249/28249 bytes (100.0%)
Transfer of 'messages.4.gz' is at 32768/71703 bytes (45.7%)
Transfer of 'messages.4.gz' is at 65536/71703 bytes (91.4%)
Transfer of 'messages.4.gz' is at 71703/71703 bytes (100.0%)
```

Again, consult the excellent `paramiko` documentation at the URL just mentioned to see the simple but complete set of file operations that SFTP supports.

Other Features

We have just covered, in the last few sections, all of the SSH operations that are supported by methods on the basic `SSHClient` object. The more obscure features that you might be familiar with—like remote X11 sessions, and port forwarding—require that you go one level deeper in the `paramiko` interface and talk directly to the client's "transport" object.

The transport is the class that actually knows the low-level operations that get combined to power an SSH connection. You can ask a client for its transport very easily:

```
>>> transport = client.get_transport()
```

Though we lack the room to cover further SSH features here, the understanding of SSH that you have gained in this chapter should help you understand them given the `paramiko` documentation combined with example code—whether from the `demos` directory of the `paramiko` project itself, or from blogs, Stack Overflow, or other materials about `paramiko` that you might find online.

One feature that we should mention explicitly is port forwarding, where SSH opens a port on either the local or remote host—at least making the port available to connections from localhost, and possibly also accepting connections from other machines on the Internet—and "forwards" these connections across the SSH channel where it connects to some other host and port on the remote end, passing data back and forth.

Port forwarding can be very useful. For example, I sometimes find myself developing a web application that I cannot run easily on my laptop because it needs access to a database and other resources that are available only out on a server farm. But I might not want the hassle of running the application on a public port—that I might have to adjust firewall rules to open—and then getting HTTPS running so that third parties cannot see my work-in-progress.

An easy solution is to run the under-development web application on the remote development machine the way I would locally—listening on `localhost:8080` so that it cannot be contacted from another computer—and then tell SSH that I want connections to my local port 8080, made here on my laptop, to be forwarded out so that they really connect to port 8080 on that local machine:

```
$ ssh -L 8080:localhost:8080 devel.example.com
```

If you need to create port-forwards when running an SSH connection with `paramiko`, then I have bad news and good news. The bad news is that the top-level `SSHClient` does not, alas, provide an easy way to create a forward like it supports more common operations like shell sessions. Instead, you will have to create the forward by talking directly to the "transport" object, and then writing loops that copy data in both directions over the forward yourself.

But the good news is that `paramiko` comes with example scripts showing exactly how to write port-forwarding loops. These two scripts, from the main `paramiko` trunk, should get you started:

```
http://github.com/robey/paramiko/blob/master/demos/forward.py
http://github.com/robey/paramiko/blob/master/demos/rforward.py
```

Of course, since the port-forward data is passed back and forth across channels inside the SSH connection, you do not have to worry if they are raw, unprotected HTTP or other traffic that is normally visible to third parties: since they are now embedded inside SSH, they are protected by its own encryption from being intercepted.

Summary

Remote-shell protocols let you connect to remove machines, run shell commands, and see their output, just like the commands were running inside a local terminal window. Sometimes you use these protocols to connect to an actual Unix shell, and sometimes to small embedded shells in routers or other networking hardware that needs configuring.

As always when talking to Unix commands, you need to be aware of output buffering, special shell characters, and terminal input buffering as issues that can make your life difficult by munging your data or even hanging your shell connection.

The Telnet protocol is natively supported by the Python Standard Library through its `telnetlib` module. Although Telnet is ancient, insecure, and can be difficult to script, it may often be the only protocol supported by simple devices to which you want to connect.

The SSH "Secure Shell" protocol is the current state of the art, not only for connecting to the command line of a remote host, but for copying files and forwarding TCP/IP ports as well. Python has quite excellent SSH support thanks to the third-party `paramiko` package. When making an SSH connection, you need to remember three things:

- `paramiko` will need to verify (or be told explicitly to ignore) the identity of the remote machine, which is defined as the host key that it presents when the connection is made.
- Authentication will typically be accomplished through a password, or through the use of a public-private key pair whose public half you have put in your `authorized_keys` file on the remote server.
- Once authenticated you can start all sorts of SSH services—remote shells, individual commands, and file-transfer sessions—and they can all run at once without your having to open new SSH connections, thanks to the fact that they will all get their own "channel" within the master SSH connection.

CHAPTER 17

FTP

The File Transfer Protocol (FTP) was once among the most widely used protocols on the Internet, invoked whenever a user wanted to transfer files between Internet-connected computers.

Alas, the protocol has seen better days; today, a better alternative exists for every one of its major roles. There were four primary activities that it once powered.

The first, and overwhelming, use of FTP was for file download. Just like people who browse the Web today, earlier generations of Internet users were able to consume far more content than they each tended to generate. Lists of "anonymous" FTP servers that allowed public access were circulated, and users connected to retrieve documents, the source code to new programs, and media like images or movies. (You logged into them with the username "anonymous" or "ftp," and then—out of politeness, so they would know who was using their bandwidth—you typed your e-mail address as the password.) And FTP was always the protocol of choice when files needed to be moved between computer accounts, since trying to transfer large files with Telnet clients was often a dicey proposition.

Second, FTP was often jury-rigged to provide for anonymous upload. Many organizations wanted outsiders to be able to submit documents or files, and their solution was to set up FTP servers that allowed files to be written into a directory whose contents could not, then, be listed back again. That way, users could not see (and hopefully could not guess!) the names of the files that other users had just submitted and get to them before the site administrators did.

Third, the protocol was often in use to support the synchronization of entire trees of files between computer accounts. By using a client that provided for "recursive" FTP operations, users could push entire directory trees from one of their accounts to another, and server administrators could clone or install new services without having to re-build them from scratch on a new machine. When using FTP like this, users were generally not aware of how the actual protocol worked, or of the many separate commands needed to transfer so many different files: instead, they hit a button and a large batch operation would run and then complete.

Fourth and finally, FTP was used for its original purpose: interactive, full-fledged file management. The early FTP clients presented a command-line prompt that felt something like a Unix shell account itself, and—as we shall see—the protocol borrows from shell accounts both the idea of a "current working directory" and of a `cd` command to move from one directory to another. Later clients mimicked the idea of a Mac-like interface, with folders and files drawn on the computer screen. But in either case, in the activity of filesystem browsing the full capabilities of FTP finally came into play: it supported not only the operations of listing directories and uploading and downloading files, but of creating and deleting directories, adjusting file permissions, and re-naming files.

What to Use Instead of FTP

Today, there are better alternatives than the FTP protocol for pretty much anything you could want to do with it. You will still occasionally see URLs that start with `ftp:`, but they are becoming quite rare. Use this chapter either because you have a legacy need to speak FTP from your Python program, or because you want to learn more about file transfer protocols in general and FTP is a good, historical place to start.

The biggest problem with the protocol is its lack of security: not only files, but usernames and passwords are sent completely in the clear and can be viewed by anyone observing network traffic.

A second issue is that an FTP user tends to make a connection, choose a working directory, and do several operations all over the same network connection. Modern Internet services, with millions of users, prefer protocols like HTTP (see Chapter 9) that consist of short, completely self-contained requests, instead of long-running FTP connections that require the server to remember things like a current working directory.

A final big issue is filesystem security. The early FTP servers, instead of showing users just a sliver of the host filesystem that the owner wanted exposed, tended to simply expose the entire filesystem, letting users `cd` to `/` and snoop around to see how the system was configured. True, you could run the server under a separate `ftp` user and try to deny that user access to as many files as possible; but many areas of the Unix filesystem need to be publicly readable simply so that normal users can use the programs there. While servers were eventually written that exposed only part of the host filesystem, this more or less violated the original intention: that an FTP session would look like a Telnet command-line prompt, even down to the fact that full pathnames were used that started at the filesystem root.

So what are the alternatives?

- For file download, HTTP (Chapter 9) is the standard protocol on today's Internet, protected with SSL when necessary for security. Instead of exposing system-specific file name conventions like FTP, HTTP supports system-independent URLs.
- Anonymous upload is a bit less standard, but the general tendency is to use a form on a web page that instructs the browser to use an HTTP POST operation to transmit the file that the user selects.
- File synchronization has improved immeasurably since the days when a recursive FTP file copy was the only common way to get files to another computer. Instead of wastefully copying every file, modern commands like `rsync` or `rdist` efficiently compare files at both ends of the connection and copy only the ones that are new or have changed. (They are not covered in this book; try Googling for them.)
- Full filesystem access is actually the one area where FTP can still commonly be found on today's Internet: thousands of cut-rate ISPs continue to support FTP, despite its insecurity, as the means by which users copy their media and (typically) PHP source code into their web account. A much better alternative today is for service providers to support SFTP instead (see Chapter 16).

■ **Note** The FTP standard is RFC959, available at `http://www.faqs.org/rfcs/rfc959.html`.

Communication Channels

FTP is unusual because, by default, it actually uses *two* TCP connections during operation. One connection is the control channel, which carries commands and the resulting acknowledgments or error codes. The second connection is the data channel, which is used solely for transmitting file data or other blocks of information, such as directory listings. Technically, the data channel is full duplex, meaning that it allows files to be transmitted in both directions simultaneously. However, in actual practice, this capability is rarely used.

In the traditional sense, the process of downloading a file from an FTP server ran mostly like this:

1. First, the FTP client establishes a command connection by connecting to the FTP port on the server.
2. The client authenticates itself, usually with username and password.
3. The client changes directory on the server to where it wants to deposit or retrieve files.
4. The client begins listening on a new port for the data connection, and then informs the server about that port.
5. The server connects to the port the client requested.
6. The file is transmitted.
7. The data connection is closed.

This worked well in the early days of the Internet; back then, most every machine that could run FTP had a public IP address, and firewalls were relatively rare. Today, however, the picture is more complicated. Firewalls blocking incoming connections to desktop and laptop machines are now quite common, and many wireless, DSL, and in-house business networks do not offer client machines real public IP addresses anyway.

To accommodate this situation, FTP also supports what is known as *passive mode*. In this scenario, the data connection is made backward: the server opens an extra port, and tells the client to make the second connection. Other than that, everything behaves the same way.

Passive mode is today the default with most FTP clients, as well as Python's `ftplib` module, which this chapter will teach you about.

Using FTP in Python

The Python module `ftplib` is the primary interface to FTP for Python programmers. It handles the details of establishing the various connections for you, and provides convenient ways to automate common commands.

■ **Tip** If you are interested only in downloading files, the `urllib2` module introduced in Chapter 1 supports FTP, and may be easier to use for simple downloading tasks; just run it with an `ftp://` URL. In this chapter, we describe `ftplib` because it provides FTP-specific features that are not available with `urllib2`.

Listing 17–1 shows a very basic `ftplib` example. The program connects to a remote server, displays the welcome message, and prints the current working directory.

***Listing 17–1.** Making a Simple FTP Connection*

```
#!/usr/bin/env python
# Basic connection - Chapter 17 - connect.py

from ftplib import FTP

f = FTP('ftp.ibiblio.org')

print "Welcome:", f.getwelcome()
```

```
f.login()
print "Current working directory:", f.pwd()
f.quit()
```

The welcome message will generally have no information that could be usefully parsed by your program, but you might want to display it if a user is calling your client interactively. The `login()` function can take several parameters, including a username, password, and a third, rarely used authentication token that FTP calls an "account." Here we have called it without parameters, which makes it log in as the user "anonymous" with a generic value for the password.

Recall that an FTP session can visit different directories, just like a shell prompt can move between locations with `cd`. Here, the `pwd()` function returns the current working directory on the remote site of the connection. Finally, the `quit()` function logs out and closes the connection.

Here is what the program outputs when run:

```
$ ./connect.py
Welcome: 220 ProFTPD Server (Bring it on...)
Current working directory: /
```

ASCII and Binary Files

When making an FTP transfer, you have to decide whether you want the file treated as a monolithic block of binary data, or whether you want it parsed as a text file so that your local machine can paste its lines back together using whatever end-of-line character is native to your platform.

A file transferred in so-called "ASCII mode" is delivered one line at a time, so that you can glue the lines back together on the local machine using its own line-ending convention. Take a look at Listing 17–2 for a Python program that downloads a well-known text file and saves it in your local directory.

Listing 17–2. *Downloading an ASCII File*

```
#!/usr/bin/env python
# ASCII download - Chapter 17 - asciidl.py
# Downloads README from remote and writes it to disk.

import os
from ftplib import FTP

if os.path.exists('README'):
»   raise IOError('refusing to overwrite your README file')

def writeline(data):
»   fd.write(data)
»   fd.write(os.linesep)
f = FTP('ftp.kernel.org')
f.login()
f.cwd('/pub/linux/kernel')

fd = open('README', 'w')
f.retrlines('RETR README', writeline)
fd.close()

f.quit()
```

In the listing, the cwd() function selects a new working directory on the remote system. Then the retrlines() function begins the transfer. Its first parameter specifies a command to run on the remote system, usually RETR, followed by a file name. Its second parameter is a function that is called, over and over again, as each line of the text file is retrieved; if omitted, the data is simply printed to standard output. The lines are passed with the end-of-line character stripped, so the homemade writeline() function simply appends your system's standard line ending to each line as it is written out.

Try running this program; there should be a file in your current directory named README after the program is done.

Basic binary file transfers work in much the same way as text-file transfers; Listing 17–3 shows an example.

Listing 17–3. *Downloading a Binary File*

```
#!/usr/bin/env python
# Binary upload - Chapter 17 - binarydl.py

import os
from ftplib import FTP

if os.path.exists('patch8.gz'):
»   raise IOError('refusing to overwrite your patch8.gz file')

f = FTP('ftp.kernel.org')
f.login()
f.cwd('/pub/linux/kernel/v1.0')

fd = open('patch8.gz', 'wb')
f.retrbinary('RETR patch8.gz', fd.write)
fd.close()

f.quit()
```

When run, it deposits a file named patch8.gz in your current working directory. The retrbinary() function simply passes blocks of data to the specified function. This is convenient, since a file object's write() function expects just such data—so in this case, no custom function is necessary.

Advanced Binary Downloading

The ftplib module provides a second function that can be used for binary downloading: ntransfercmd(). This command provides a lower-level interface, but can be useful if you want to know a little bit more about what's going on during the download.

In particular, this more advanced command lets you keep track of the number of bytes transferred, and you can use that information to display status updates for the user. Listing 17–4 shows a sample program that uses ntransfercmd().

Listing 17–4. *Binary Download with Status Updates*

```
#!/usr/bin/env python
# Advanced binary download - Chapter 17 - advbinarydl.py

import os, sys
from ftplib import FTP
```

```
if os.path.exists('linux-1.0.tar.gz'):
»   raise IOError('refusing to overwrite your linux-1.0.tar.gz file')

f = FTP('ftp.kernel.org')
f.login()

f.cwd('/pub/linux/kernel/v1.0')
f.voidcmd("TYPE I")

datasock, size = f.ntransfercmd("RETR linux-1.0.tar.gz")
bytes_so_far = 0
fd = open('linux-1.0.tar.gz', 'wb')

while 1:
»   buf = datasock.recv(2048)
»   if not buf:
»   »   break
»   fd.write(buf)
»   bytes_so_far += len(buf)
»   print "\rReceived", bytes_so_far,
»   if size:
»   »   print "of %d total bytes (%.1f%%)" % (
»   »   »   size, 100 * bytes_so_far / float(size)),
»   else:
»   »   print "bytes",
»   sys.stdout.flush()

print
fd.close()
datasock.close()
f.voidresp()
f.quit()
```

There are a few new things to note here. First comes the call to `voidcmd()`. This passes an FTP command directly to the server, checks for an error, but returns nothing. In this case, the raw command is `TYPE I`. That sets the transfer mode to "image," which is how FTP refers internally to binary files. In the previous example, `retrbinary()` automatically ran this command behind the scenes, but the lower-level `ntransfercmd()` does not.

Next, note that `ntransfercmd()` returns a tuple consisting of a data socket and an estimated size. *Always* bear in mind that the size is merely an *estimate,* and should not be considered authoritative; the file may end sooner, or it might go on much longer, than this value. Also, if a size estimate from the FTP server is simply not available, then the estimated size returned will be `None`.

The object `datasock` is, in fact, a plain TCP socket, which has all of the behaviors described in the first section of this book (see Chapter 3 in particular). In this example, a simple loop calls `recv()` until it has read all of the data from the socket, writing it out to disk along the way and printing out status updates to the screen.

■ **Tip** Notice two things about the status updates printed to the screen by Listing 17–4, by the way. First, rather than printing a scrolling list of lines that disappear out of the top of the terminal, we begin each line with a carriage return `'\r'`, which moves the cursor back to your terminal's left edge so that each status line overwrites the previous one and creates the illusion of an increasing, animated percentage. Second, because we are ending each `print` statement with a comma and are never actually letting them finish a line of output, we have to `flush()` the standard output to make sure that the status updates immediately reach the screen.

After receiving the data, it is important to close the data socket and call `voidresp()`, which reads the command response code from the server, raising an exception if there was any error during transmission. Even if you do not care about detecting errors, failing to call `voidresp()` will make future commands likely to fail because the server's output socket will be blocked waiting for you to read the results.

Here is an example of running this program:

```
$ ./advbinarydl.py
Received 1259161 of 1259161 bytes (100.0%)
```

Uploading Data

File data can also be uploaded through FTP. As with downloading, there are two basic functions for uploading: `storbinary()` and `storlines()`. Both take a command to run, and a file-like object to transmit. The `storbinary()` function will call the `read()` method repeatedly on that object until its content is exhausted, while `storlines()`, by contrast, calls the `readline()` method.

Unlike the corresponding download functions, these methods do not require you to provide a callable function of your own. (But you could, of course, pass a file-like object of your own crafting whose `read()` or `readline()` method computes the outgoing data as the transmission proceeds!)

Listing 17–5 shows how to upload a file in binary mode.

Listing 17–5. *Binary Upload*

```
#!/usr/bin/env python
# Binary download - Chapter 17 - binaryul.py

from ftplib import FTP
import sys, getpass, os.path

if len(sys.argv) != 5:
»   print "usage: %s <host> <username> <localfile> <remotedir>" % (
»   »   sys.argv[0])
»   exit(2)

host, username, localfile, remotedir = sys.argv[1:]
password = getpass.getpass(
»   "Enter password for %s on %s: " % (username, host))
```

```
f = FTP(host)
f.login(username, password)
f.cwd(remotedir)

fd = open(localfile, 'rb')
f.storbinary('STOR %s' % os.path.basename(localfile), fd)
fd.close()

f.quit()
```

This program looks quite similar to our earlier efforts. Since most anonymous FTP sites do not permit file uploading, you will have to find a server somewhere to test it against; I simply installed the old, venerable `ftpd` on my laptop for a few minutes and ran the test like this:

```
$ python binaryul.py localhost brandon test.txt /tmp
```

I entered my password at the prompt (`brandon` is my username on this machine). When the program finished, I checked and, sure enough, a copy of the `test.txt` file was now sitting in `/tmp`. Remember *not* to try this over a network to another machine, since FTP does not encrypt or protect your password!

You can modify this program to upload a file in ASCII mode by simply changing `storbinary()` to `storlines()`.

Advanced Binary Uploading

Just like the download process had a complicated raw version, it is also possible to upload files "by hand" using `ntransfercmd()`, as shown in Listing 17–6.

Listing 17–6. *Uploading Files a Block at a Time*

```
#!/usr/bin/env python
# Advanced binary upload - Chapter 17 - advbinaryul.py

from ftplib import FTP
import sys, getpass, os.path

BLOCKSIZE = 8192  # chunk size to read and transmit: 8 kB

if len(sys.argv) != 5:
    print "usage: %s <host> <username> <localfile> <remotedir>" % (
        sys.argv[0])
    exit(2)

host, username, localfile, remotedir = sys.argv[1:]
password = getpass.getpass("Enter password for %s on %s: " % \
        (username, host))
f = FTP(host)
f.login(username, password)
f.cwd(remotedir)
f.voidcmd("TYPE I")
fd = open(localfile, 'rb')
datasock, esize = f.ntransfercmd('STOR %s' % os.path.basename(localfile))
size = os.stat(localfile)[6]
bytes_so_far = 0
```

```
while 1:
»   buf = fd.read(BLOCKSIZE)
»   if not buf:
»   »   break
»   datasock.sendall(buf)
»   bytes_so_far += len(buf)
»   print "\rSent", bytes_so_far, "of", size, "bytes", \
»   »   "(%.1f%%)\r" % (100 * bytes_so_far / float(size))
»   sys.stdout.flush()

print
datasock.close()
fd.close()
f.voidresp()
f.quit()
```

Note that the first thing we do when finished with our transfer is to call `datasock.close()`. When uploading data, closing the socket is the signal to the server that the upload is complete! If you fail to close the data socket after uploading all your data, the server will keep waiting for the rest of the data to arrive.

Now we can perform an upload that continuously displays its status as it progresses:

```
$ python binaryul.py localhost brandon patch8.gz /tmp
Enter password for brandon on localhost:
Sent 6408 of 6408 bytes (100.0%)
```

Handling Errors

Like most Python modules, `ftplib` will raise an exception when an error occurs. It defines several exceptions of its own, and it can also raise `socket.error` and `IOError`. As a convenience, it offers a tuple, named `ftplib.all_errors`, that lists all of the exceptions that can possibly be raised by `ftplib`. This is often a useful shortcut for writing a `try…except` clause.

One of the problems with the basic `retrbinary()` function is that, in order to use it easily, you will usually wind up opening the file on the local end before beginning the transfer on the remote side. If your command aimed at the remote side retorts that the file does not exist, or if the `RETR` command otherwise fails, then you will have to close and delete the local file you have just created (or else wind up littering the filesystem with zero-length files).

With the `ntransfercmd()` method, by contrast, you can check for a problem prior to opening a local file. Listing 17–6 already follows these guidelines: if `ntransfercmd()` fails, the exception will cause the program to terminate before the local file is opened.

Scanning DirectoriesFTP provides two ways to discover information about server files and directories. These are implemented in `ftplib` as the `nlst()` and `dir()` methods.

The `nlst()` method returns a list of entries in a given directory—all of the files and directories inside. However, the bare names are all that is returned. There is no other information about which particular entries are files or are directories, on the sizes of the files present, or anything else.

The more powerful `dir()` function returns a directory listing from the remote. This listing is in a system-defined format, but typically contains a file name, size, modification date, and file type. On UNIX servers, it is typically the output of one of these two shell commands:

```
$ ls -l
$ ls -la
```

Windows servers may use the output of `dir`. While the output may be useful to an end user, it is difficult for a program to use, due to the varying output formats. Some clients that need this data implement parsers for the many different formats that `ls` and `dir` produce across machines and operating system versions; others can only parse the one format in use in a particular situation.

Listing 17–7 shows an example of using `nlst()` to get directory information.

Listing 17–7. *Getting a Bare Directory Listing*

```
#!/usr/bin/env python
# NLST example - Chapter 17 - nlst.py

from ftplib import FTP

f = FTP('ftp.ibiblio.org')
f.login()
f.cwd('/pub/academic/astronomy/')
entries = f.nlst()
entries.sort()
print len(entries), "entries:"
for entry in entries:
»   print entry
f.quit()
```

Listing 17–7 shows an example of using `nlst()` to get directory information. When you run this program, you will see output like this:

```
$ python nlst.py
13 entries:
INDEX
README
ephem_4.28.tar.Z
hawaii_scope
incoming
jupitor-moons.shar.Z
lunar.c.Z
lunisolar.shar.Z
moon.shar.Z
planetary
sat-track.tar.Z
stars.tar.Z
xephem.tar.Z
```

If you were to use an FTP client to manually log on to the server, you would see the same files listed. Notice that the file names are in a convenient format for automated processing—a bare list of file names—but that there is no extra information. The result will be different when we try another file listing command in Listing 17–8.

Listing 17–8. *Getting a Fancy Directory Listing*

```
#!/usr/bin/env python
# dir() example - Chapter 17 - dir.py

from ftplib import FTP

f = FTP('ftp.ibiblio.org')
```

```
f.login()
f.cwd('/pub/academic/astronomy/')
entries = []
f.dir(entries.append)
print "%d entries:" % len(entries)
for entry in entries:
    print entry
f.quit()
```

Notice that the filenames are in a convenient format for automated processing — a bare list of filenames — but that is no extra information. Contrast the bare list of file names we saw earlier with the output from Listing 17–8, which uses `dir()`:

```
$ python dir.py
13 entries:
-rw-r--r--   1 (?)      (?)            750 Feb 14  1994 INDEX
-rw-r--r--   1 root     bin            135 Feb 11  1999 README
-rw-r--r--   1 (?)      (?)         341303 Oct  2  1992 ephem_4.28.tar.Z
drwxr-xr-x   2 (?)      (?)           4096 Feb 11  1999 hawaii_scope
drwxr-xr-x   2 (?)      (?)           4096 Feb 11  1999 incoming
-rw-r--r--   1 (?)      (?)           5983 Oct  2  1992 jupitor-moons.shar.Z
-rw-r--r--   1 (?)      (?)           1751 Oct  2  1992 lunar.c.Z
-rw-r--r--   1 (?)      (?)           8078 Oct  2  1992 lunisolar.shar.Z
-rw-r--r--   1 (?)      (?)          64209 Oct  2  1992 moon.shar.Z
drwxr-xr-x   2 (?)      (?)           4096 Jan  6  1993 planetary
-rw-r--r--   1 (?)      (?)         129969 Oct  2  1992 sat-track.tar.Z
-rw-r--r--   1 (?)      (?)          16504 Oct  2  1992 stars.tar.Z
-rw-r--r--   1 (?)      (?)         410650 Oct  2  1992 xephem.tar.Z
```

The `dir()` method takes a function that it calls for each line, delivering the directory listing in pieces just like `retrlines()` delivers the contents of particular files. Here, we simply supply the `append()` method of our plain old Python `entries` list.

Detecting Directories and Recursive Download

If you cannot guarantee what information an FTP server might choose to return from its `dir()` command, how are you going to tell directories from normal files—an essential step to downloading entire trees of files from the server?

The answer, shown in Listing 17–9, is to simply try a `cwd()` into every name that `nlst()` returns and, if you succeed, conclude that the entity is a directory! This sample program does not do any actual downloading; instead, to keep things simple (and not flood your disk with sample data), it simply prints out the directories it visits to the screen.

Listing 17–9. *Trying to Recurse into Directories*

```
#!/usr/bin/env python
# Recursive downloader - Chapter 17 - recursedl.py

import os, sys
from ftplib import FTP, error_perm

def walk_dir(f, dirpath):
    original_dir = f.pwd()
```

```
»   try:
»   »   f.cwd(dirpath)
»   except error_perm:
»   »   return  # ignore non-directories and ones we cannot enter
»   print dirpath
»   names = f.nlst()
»   for name in names:
»   »   walk_dir(f, dirpath + '/' + name)
»   f.cwd(original_dir)  # return to cwd of our caller

f = FTP('ftp.kernel.org')
f.login()
walk_dir(f, '/pub/linux/kernel/Historic/old-versions')
f.quit()
```

This sample program will run a bit slow—there are, it turns out, quite a few files in the old-versions directory on the Linux Kernel Archive—but within a few dozen seconds, you should see the resulting directory tree displayed on the screen:

```
$ python recursedl.py
/pub/linux/kernel/Historic/old-versions
/pub/linux/kernel/Historic/old-versions/impure
/pub/linux/kernel/Historic/old-versions/old
/pub/linux/kernel/Historic/old-versions/old/corrupt
/pub/linux/kernel/Historic/old-versions/tytso
```

By adding a few `print` statements, you could supplement this list of directories by displaying every one of the files that the recursive process is (slowly) discovering. And by adding another few lines of code, you could be downloading the files themselves to corresponding directories that you create locally. But the only really essential logic for a recursive download is already operating in Listing 17–9: the only foolproof way to know if an entry is a directory that you are allowed to enter is to try running `cwd()` against it.

Creating Directories, Deleting Things

Finally, FTP supports file deletion, and supports both the creation and deletion of directories. These more obscure calls are all described in the `ftplib` documentation:

- `delete(filename)` will delete a file from the server.
- `mkd(dirname)` attempts to create a new directory.
- `rmd(dirname)` will delete a directory; note that most systems require the directory to be empty first.
- `rename(oldname, newname)` works, essentially, like the Unix command `mv`: if both names are in the same directory, the file is essentially re-named; but if the destination specifies a name in a different directory, then the file is actually moved.

Note that these commands, like all other FTP operations, are performed more or less as though you were really logged on to the remote server command line as the same username with which you logged into FTP—and the chances of your having permission to manipulate files on a given server are lower than being able to download files, or even to create new files in an upload directory. Still, it is because of

these last few commands that FTP can be used to back file-browser applications that let users drag and drop files and directories seamlessly between their local system and the remote host.

Doing FTP Securely

Though we noted at the beginning of this chapter that there are far better protocols to adopt than FTP for pretty much anything you could use FTP to accomplish—in particular the robust and secure SFTP extension to SSH (see Chapter 16)—we should be fair and note that some few FTP servers support TLS encryption (see Chapter 6) and that Python's `ftplib` does provide this protection if you want to take advantage of it.

To use TLS, create your FTP connection with the `FTP_TLS` class instead of the plain `FTP` class; simply by doing this, your username and password and, in fact, the entire FTP command channel will be protected from prying eyes. If you then additionally run the class's `prot_p()` method (it takes no arguments), then the FTP data connection will be protected as well. Should you for some reason want to return to using an un-encrypted data connection during the session, there is a `prot_c()` method that returns the data stream to normal. Again, your commands will continue to be protected as long as you are using the `FTP_TLS` class.

Check the Python Standard Library documentation for more details (they include a small code sample) if you wind up needing this extension to FTP:
`http://docs.python.org/library/ftplib.html#ftplib.FTP_TLS`

Summary

FTP lets you transfer files between a client running on your machine and a remote FTP server. Though the protocol is insecure and outdated when compared to better choices like SFTP, you might still find services and machines that require you to use it. In Python, the `ftplib` library is used to talk to FTP servers.

FTP supports binary and ASCII transfers. ASCII transfers are usually used for text files, and permit line endings to be adjusted as the file is transferred. Binary transfers are used for everything else. The `retrlines()` function is used to download a file in ASCII mode, while `retrbinary()` downloads a file in binary mode.

You can also upload files to a remote server. The `storlines()` function uploads a file in ASCII mode, and `storbinary()` uploads a file in binary mode.

The `ntransfercmd()` function can be used for binary uploads and downloads. It gives you more control over the transfer process and is often used to support a progress bar for the user.

The `ftplib` module raises exceptions on errors. The special tuple `ftplib.all_errors` can be used to catch any error that it might raise.

You can use `cwd()` to change to a particular directory on the remote end. The `nlst()` command returns a simple list of all entries (files or directories) in a given directory. The `dir()` command returns a more detailed list but in server-specific format. Even with only `nlst()`, you can usually detect whether an entry is a file or directory by attempting to use `cwd()` to change to it and noting whether you get an error.

CHAPTER 18

RPC

Remote Procedure Call (RPC) systems let you call a remote function using the same syntax that you would use when calling a routine in a local API or library. This tends to be useful in two situations:

- Your program has a lot of work to do, and you want to spread it across several machines by making calls across the network.
- You need data or information that is only available on another hard drive or network, and an RPC interface lets you easily send queries to another system to get back an answer.

The first remote procedure systems tended to be written for low-level languages like C, and therefore placed bytes on the network that looked very much like the bytes already being written on to the processor stack every time one C function called another. And just as a C program could not safely call a library function without a header file that told it exactly how to lay out the function's arguments in memory (any errors often resulted in a crash), RPC calls could not be made without knowing ahead of time how the data would be serialized. Each RPC payload, in fact, looked exactly like a block of binary data that has been formatted by the Python `struct` module that we looked at in Chapter 5.

But today our machines and networks are fast enough that we are often in the mood to exchange some memory and speed for protocols that are more robust and that require less coordination between two pieces of code that are in conversation. Older RPC protocols would have sent a stream of bytes like the following:

```
0, 0, 0, 1, 64, 36, 0, 0, 0, 0, 0, 0
```

It would have been up to the receiver to know that the function's parameters are a 32-bit integer and a 64-bit floating point number, and then to decode the twelve bytes to the integer 1 and the number 10.0. But these days the payload is likely to be XML, written in a way that makes it all but impossible to interpret the arguments as anything other than an integer and a floating-point number:

```
<params>
  <param><value><i4>41</i4></value></param>
  <param><value><double>10.</double></value></param>
</params>
```

Our forefathers would be appalled that twelve bytes of actual binary data have bloated into 108 bytes of protocol that has to be generated by the sender and then parsed on the receiving end, consuming hundreds of CPU cycles. But the elimination of ambiguity in our protocols has generally been considered worth the expense. Of course, this pair of arguments can be expressed with less verbosity by using a more modern payload format like JSON:

```
[1, 10.0]
```

But in both cases you can see that unambiguous textual representation has become the order of the day, and it has replaced the older practice of sending raw binary data whose meaning had to be known in advance.

Of course, you might be asking by this point exactly what makes RPC protocols at all special. After all, the choices we are talking about here — that you have to choose a data format, send a request, and receive a response in return — are not peculiar to procedure calls; they are common to any meaningful network protocol whatsoever! Both HTTP and SMTP, to take two examples from previous chapters, have to serialize data and define message formats. So again, you might wonder: what makes RPC at all special?

There are three features that mark a protocol as an example of RPC.

First, an RPC protocol is distinguished by lacking strong semantics for the meaning of each call. Whereas HTTP is used to retrieve documents, and SMTP supports the delivery of messages, an RPC protocol does not assign any meaning to the data passed except to support basic data types like integers, floats, strings, and lists. It is instead up to each particular API that you fashion using an RPC protocol to define what its calls mean.

Second, RPC mechanisms are a way to invoke methods, but they do not define them. When you read the specification of a more single-purpose protocol like HTTP or SMTP, you will note that they define a finite number of basic operations — like GET and PUT in the case of HTTP, or EHLO and MAIL when you are using SMTP. But RPC mechanisms leave it up to you to define the *verbs* or function calls that your server will support; they do not limit them in advance.

Third, when you use RPC, your client and server code should not look very different from any other code that uses function calls. Unless you know that an object represents a remote server, the only pattern you might notice in the code is a certain caution with respect to the objects that are passed — lots of numbers and strings and lists, but not *live* objects like open files. But while the kinds of arguments passed might be limited, the function calls will "look normal" and not require decoration or elaboration in order to pass over the network.

Features of RPC

Besides serving their the essential purpose of letting you make what appear to be local function or method calls that are in fact passing across the network to a different server, RPC protocols have several key features, and also some differences, that you should keep in mind when choosing and then deploying an RPC client or server.

First, every RPC mechanism has limits on the kind of data you can pass. The most general-purpose RPC mechanisms tend to be the most restrictive because they are designed to work with many different programming languages and can only support lowest-common-denominator features that appear in almost all programming languages.

The most popular protocols, therefore, support only a few kinds of numbers and strings; one sequence or list data type; and then something like a struct or associative array. Many Python programmers are disappointed to learn that only positional arguments are typically supported, since so few other languages at this point support keyword arguments.

When an RPC mechanism is tied to a specific programming language, it is free to support a wider range of parameters; and in some cases, even live objects can be passed if the protocol can figure out some way to rebuild them on the remote side. In this case, only objects backed by live operating system resources — like an open file, live socket, or area of shared memory — become impossible to pass over the network.

A second common feature is the ability of the server to signal that an exception occurred while it was running the remote function. In such cases, the client RPC library will typically raise an exception itself to tell the client that something has gone wrong. Of course, live traceback information of the sort that Python programmers are often fond of using typically cannot be passed back; each stack frame, for example, would try to refer to modules that do not actually exist in the client program. But at least some

sort of proxy exception that gives the right error message must be raised on the client side of the RPC conversation when a call fails on the server.

Third, many RPC mechanisms provide introspection, which is a way for clients to list the calls that are supported and perhaps to discover what arguments they take. Some heavyweight RPC protocols actually require the client and server to exchange large documents describing the library or API they support; others just let the list of function names and argument types be fetched by the client from the server; and other RPC implementations support no introspection at all. Python tends to be a bit weak in supporting introspection because Python, unlike a statically-typed language, does not know what argument types are intended by the programmer who has written each function.

Fourth, each RPC mechanism needs to support some addressing scheme whereby you can reach out and connect to a particular remote API. Some such mechanisms are quite complicated, and they might even have the ability to automatically connect you to the correct server on your network for performing a particular task, without your having to know its name beforehand. Other mechanisms are quite simple and just ask you for the IP address, port number, or URL of the service you want to access. These mechanisms expose the underlying network addressing scheme, rather than creating a scheme of their own.

Finally, some RPC mechanisms support authentication, access control, and even full impersonation of particular user accounts when RPC calls are made by several different client programs wielding different credentials. But features like these are not always available; and, in fact, simple and popular RPC mechanisms usually lack them entirely. Often these RPC schemes use an underlying protocol like HTTP that provides its own authentication, and they leave it up to you to configure whatever passwords, public keys, or firewall rules are necessary to secure the lower-level protocol if you want your RPC service protected from arbitrary access.

XML-RPC

We will begin our brief tour of RPC mechanisms by looking at the facilities built into Python for speaking XML-RPC. This might seem like a poor choice for our first example. After all, XML is famously clunky and verbose, and the popularity of XML-RPC in new services has been declining for years.

But XML-RPC has native support in Python precisely because it was one of the first RPC protocols of the Internet age, operating natively over HTTP instead of insisting on its own on-the-wire protocol. This means our examples will not even require any third-party modules. While we will see that this makes our RPC server somewhat less capable than if we moved to a third-party library, this will also make the examples good ones for an initial foray into RPC.

THE XML-RPC PROTOCOL

Purpose: **Remote procedure calls**

Standard: **www.xmlrpc.com/spec**

Runs atop: **HTTP**

Data types: `int`; `float`; `unicode`; `list`; `dict` with `unicode` keys; with non-standard extensions, `datetime` and `None`

Libraries: `xmlrpclib`, `SimpleXMLRPCServer`, `DocXMLRPCServer`

If you have ever used raw XML, then you are familiar with the fact that it lacks any data-type semantics; it cannot represent numbers, for example, but only *elements* that contain other elements, text strings, and text-string attributes. Thus the XML-RPC specification has to build additional semantics on top of the plain XML document format in order to specify how things like numbers should look when converted into marked-up text.

The Python Standard Library makes it easy to write either an XML-RPC client or server, though more power is available when writing a client. For example, the client library supports HTTP basic authentication, while the server does not support this. Therefore, we will begin at the simple end, with the server.

Listing 18–1 shows a basic server that starts a web server on port 7001 and listens for incoming Internet connections.

Listing 18–1. *An XML-RPC Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - xmlrpc_server.py
# XML-RPC server

import operator, math
from SimpleXMLRPCServer import SimpleXMLRPCServer

def addtogether(*things):
    """Add together everything in the list `things`."""
    return reduce(operator.add, things)

def quadratic(a, b, c):
    """Determine `x` values satisfying: `a` * x*x + `b` * x + c == 0"""
    b24ac = math.sqrt(b*b - 4.0*a*c)
    return list(set([ (-b-b24ac) / 2.0*a,
                      (-b+b24ac) / 2.0*a ]))

def remote_repr(arg):
    """Return the `repr()` rendering of the supplied `arg`."""
    return arg

server = SimpleXMLRPCServer(('127.0.0.1', 7001))
server.register_introspection_functions()
server.register_multicall_functions()
server.register_function(addtogether)
server.register_function(quadratic)
server.register_function(remote_repr)
print "Server ready"
server.serve_forever()
```

An XML-RPC service lives at a single URL of a web site, so you do not have to actually dedicate an entire port to an RPC service like this; instead, you can integrate it with a normal web application that offers all sorts of other pages, or even entire other RPC services, at other URLs. But if you do have an entire port to spare, then the Python XML-RPC server offers an easy way to bring up a web server that does nothing but talk XML-RPC.

You can see that the three sample functions that the server offers over XML-RPC — the ones that are added to the RPC service through the `register_function()` calls — are quite typical Python functions. And that, again, is the whole point of XML-RPC: it lets you make routines available for invocation over the network without having to write them any differently than if they were normal functions offered inside of your program.

The SimpleXMLRPCServer offered by the Standard Library is, as its name implies, quite simple; it cannot offer other web pages, it does not understand any kind of HTTP authentication, and you cannot ask it to offer TLS security without subclassing it yourself and adding more code. But it will serve our purposes admirably, showing you some of the basic features and limits of RPC, while also letting you get up and running in only a few lines of code.

Note that two additional configuration calls are made in addition to the three calls that register our functions. Each of them turns on an additional service that is optional, but often provided by XML-RPC servers: an introspection routine that a client can use to ask which RPC calls are supported by a given server; and the ability to support a *multicall* function that lets several individual function calls be bundled together into a single network round-trip.

This server will need to be running before we can try any of the next three program listings, so bring up a command window and get it started:

```
$ python xmlrpc_server.py
Server ready
```

The server is now waiting for connections on localhost port 7001. All of the normal addressing rules apply to this TCP server that you learned in Chapters 2 and 3, so you will have to connect to it from another command prompt on the same system. Begin by opening another command window and get ready to try out the next three listings as we review them.

First, we will try out the introspection capability that we turned on in this particular server. Note that this ability is optional, and it may not be available on many other XML-RPC services that you use online or that you deploy yourself. Listing 18–2 shows how introspection happens from the client's point of view.

Listing 18–2. *Asking an XML-RPC Server What Functions It Supports*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - xmlrpc_introspect.py
# XML-RPC client

import xmlrpclib
proxy = xmlrpclib.ServerProxy('http://127.0.0.1:7001')

print 'Here are the functions supported by this server:'
for method_name in proxy.system.listMethods():
»   if method_name.startswith('system.'):
»   »   continue
»   signatures = proxy.system.methodSignature(method_name)
»   if isinstance(signatures, list) and signatures:
»   »   for signature in signatures:
»   »   »   print '%s(%s)' % (method_name, signature)
»   else:
»   »   print '%s(...)' % (method_name,)

»   method_help = proxy.system.methodHelp(method_name)
»   if method_help:
»   »   print '  ', method_help
```

The introspection mechanism is an optional extension that is not actually defined in the XML-RPC specification itself. The client is able to call a series of special methods that all begin with the string system. to distinguish them from normal methods. These special methods give information about the other calls available. We start by calling listMethods(). If introspection is supported at all, then we will receive back a list of other method names; for this example listing, we ignore the system methods and only proceed to print out information about the other ones. For each method, we attempt to retrieve its

signature to learn what arguments and data types it accepts. Because our server is Python-based, it does not actually know what data types the functions take:

```
$ python xmlrpc_introspect.py
Here are the functions supported by this server:
concatenate(...)
   Add together everything in the list `things`.
quadratic(...)
   Determine `x` values satisfying: `a` * x*x + `b` * x + c == 0
remote_repr(...)
   Return the `repr()` rendering of the supplied `arg`.
```

However, you can see that, while parameter types are not given in this case, documentation strings are indeed provided — in fact, the `SimpleXMLRPCServer` has fetched our function's docstrings and returned them for viewing by the RPC client. There are two uses that you might find for introspection in a real-world client. First, if you are writing a program that uses a particular XML-RPC service, then its online documentation might provide human-readable help to you. Second, if you are writing a client that is hitting a series of similar XML-RPC services that vary in the methods they provide, then a `listMethods()` call might help you work out which servers offer which commands.

You will recall that the whole point of an RPC service is to make function calls in a target language look as natural as possible. And as you can see in Listing 18–3, the Standard Library's `xmlrpclib` gives you a *proxy* object for making function calls against the server. These calls look exactly like local function calls.

Listing 18–3. *Making XML-RPC Calls*

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Foundations of Python Network Programming - Chapter 18 - xmlrpc_client.py
# XML-RPC client

import xmlrpclib
proxy = xmlrpclib.ServerProxy('http://127.0.0.1:7001')
print proxy.addtogether('x', 'ÿ', 'z')
print proxy.addtogether(20, 30, 4, 1)
print proxy.quadratic(2, -4, 0)
print proxy.quadratic(1, 2, 1)
print proxy.remote_repr((1, 2.0, 'three'))
print proxy.remote_repr([1, 2.0, 'three'])
print proxy.remote_repr({'name': 'Arthur', 'data': {'age': 42, 'sex': 'M'}})
print proxy.quadratic(1, 0, 1)
```

Running the preceding code against our example server produces output from which we can learn several things about XML-RPC in particular, and RPC mechanisms in general. Note how almost all of the calls work without a hitch, and how both of the calls in this listing and the functions themselves back in Listing 18–1 look like completely normal Python; there is with nothing about them that is particular to a network:

```
$ python xmlrpc_client.py
xÿz
55
[0.0, 8.0]
[-1.0]
[1, 2.0, 'three']
[1, 2.0, 'three']
{'data': {'age': [42], 'sex': 'M'}, 'name': 'Arthur'}
```

```
Traceback (most recent call last):
  ...
xmlrpclib.Fault: <Fault 1: "<type 'exceptions.ValueError'>:math domain error">
```

The preceding snippet illustrates several key points about using XML-RPC. First, note that XML-RPC is not imposing any restrictions upon the argument types we are supplying. We can call `addtogether()` with either strings or numbers, and we can supply any number of arguments. The protocol itself does not care; it has no pre-conceived notion of how many arguments a function should take or what its types should be. Of course, if we were making calls to a language that did care — or even to a Python function that did not support variable-length argument lists — then the remote language could raise an exception. But that would be the language complaining, not the XML-RPC protocol itself.

Second, note that XML-RPC function calls, like those of Python and many other languages in its lineage, can take several arguments, but can only return a single result value. That value might be a complex data structure, but it will be returned as a single result. And the protocol does not care whether that result has a consistent shape or size; the list returned by `quadratic()` (yes, I was tired of all of the simple `add()` and `subtract()` math functions that tend to get used in XML-RPC examples!) varies in its number of elements returned without any complaint from the network logic.

Third, note that the rich variety of Python data types must be reduced to the smaller set that XML-RPC itself happens to support. In particular, XML-RPC only supports a single sequence type: the list. So when we supply `remote_repr()` with a tuple of three items, it is actually a list of three items that gets received at the server instead. This is a common feature of all RPC mechanisms when they are coupled with a particular language; types they do not directly support either have to be mapped to a different data structure (as our tuple was here turned into a list), or an exception has to be raised complaining that a particular argument type cannot be transmitted.

Fourth, complex data structures in XML-RPC can be recursive; you are not restricted to arguments that have only one level of complex data type inside. Passing a dictionary with another dictionary as one of its values works just fine, as you can see.

Finally, note that — as promised earlier — an exception in our function on the server made it successfully back across the network and was represented locally on the client by an `xmlrpclib.Fault` instance. This instance provided the remote exception name and the error message associated with it. Whatever the language used to implement the server routines, you can always expect XML-RPC exceptions to have this structure. The traceback is not terribly informative; while it tells us which call in our code triggered the exception, the innermost levels of the stack are simply the code of the `xmlrpclib` itself.

Thus far we have covered the general features and restrictions of XML-RPC. If you consult the documentation for either the client or the server module in the Standard Library, you can learn about a few more features. In particular, you can learn how to use TLS and authentication by supplying more arguments to the `ServerProxy` class. But one feature is important enough to go ahead and cover here: the ability to make several calls in a network round-trip when the server supports it (it is another one of those optional extensions), as shown in Listing 18–4.

Listing 18–4. *Using XML-RPC Multicall*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - xmlrpc_multicall.py
# XML-RPC client performing a multicall

import xmlrpclib
proxy = xmlrpclib.ServerProxy('http://127.0.0.1:7001')
multicall = xmlrpclib.MultiCall(proxy)
multicall.addtogether('a', 'b', 'c')
multicall.quadratic(2, -4, 0)
multicall.remote_repr([1, 2.0, 'three'])
for answer in multicall():
»   print answer
```

When you run this script, you can carefully watch the server's command window to confirm that only a single HTTP request is made in order to answer all three function calls that get made:

```
localhost - - [04/Oct/2010 00:16:19] "POST /RPC2 HTTP/1.0" 200 -
```

The ability to log messages like the preceding one can be turned off, by the way; such logging is controlled by one of the options in `SimpleXMLRPCServer`. Note that the default URL used by both the server and client is the path `/RPC2`, unless you consult the documentation and configure the client and server differently.

Three final points are worth mentioning before we move on to examining another RPC mechanism:

- There are two additional data types that sometimes prove hard to live without, so many XML-RPC mechanisms support them: dates and the value that Python calls `None` (other languages call this `null` or `nil` instead). Python's client and server both support options that will enable the transmission and reception of these non-standard types.
- Keyword arguments are, alas, not supported by XML-RPC, because few languages are sophisticated enough to include them and XML-RPC wants to interoperate with those languages. Some services get around this by allowing a dictionary to be passed as a function's final argument — or by disposing of positional arguments altogether and using a single dictionary argument for every function that supplies all of its parameters by name.
- Finally, keep in mind that dictionaries can only be passed if all of their keys are strings, whether normal or Unicode. See the "Self-documenting Data" section later in this chapter for more information on how to think about this restriction.

While the entire point of an RPC protocol like XML-RPC is to let you forget about the details of network transmission and focus on normal programming, you should see what your calls will look like on the wire at least once! Here is the first call to `quadratic()` that our sample client program makes:

```
<?xml version='1.0'?>
<methodCall>
<methodName>quadratic</methodName>
<params>
<param>
<value><int>2</int></value>
</param>
<param>
<value><int>-4</int></value>
</param>
<param>
<value><int>0</int></value>
</param>
</params>
</methodCall>
```

The response to the preceding call looks like this:

```
<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><array><data>
<value><double>0.0</double></value>
```

```
<value><double>8.0</double></value>
</data></array></value>
</param>
</params>
</methodResponse>
```


If this response looks a bit verbose for the amount of data that it is transmitting, then you will be happy to learn about the RPC mechanism that we tackle next.

JSON-RPC

The bright idea behind JSON is to serialize data structures to strings that use the syntax of the JavaScript programming language. This means that JSON strings can be turned back into data in a web browser simply by using the `eval()` function. By using a syntax specifically designed for data rather than adapting a verbose document markup language like XML, this remote procedure call mechanism can make your data much more compact while simultaneously simplifying your parsers and library code.

THE JSON-RPC PROTOCOL

Purpose: **Remote procedure calls**

Standard: http://json-rpc.org/wiki/specification

Runs atop: **HTTP**

Data types: `int`; `float`; `unicode`; `list`; `dict` with `unicode` keys; `None`

Libraries: many third-party, including `lovely.jsonrpc`

JSON-RPC is not supported in the Python Standard Library (at least at the time of writing), so you will have to choose one of the several third-party distributions available. You can find these distributions on the Python Package Index. My own favorite is `lovely.jsonrpc`, contributed to the Python community by Lovely Systems GmBH in Austria. If you install it in a virtual environment (see Chapter 1), then you can try out the server and client shown in Listings 18–5 and 18–6.

Listing 18–5. *A JSON-RPC Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - jsonrpc_server.py
# JSON-RPC server

from wsgiref.simple_server import make_server
import lovely.jsonrpc.dispatcher, lovely.jsonrpc.wsgi

def lengths(*args):
»   results = []
»   for arg in args:
»   »   try:
»   »   »   arglen = len(arg)
»   »   except TypeError:
»   »   »   arglen = None
»   »   results.append((arglen, arg))
»   return results
```

```
dispatcher = lovely.jsonrpc.dispatcher.JSONRPCDispatcher()
dispatcher.register_method(lengths)
app = lovely.jsonrpc.wsgi.WSGIJSONRPCApplication({'': dispatcher})
server = make_server('localhost', 7002, app)
print "Starting server"
while True:
»   server.handle_request()
```

You can see that the Lovely Systems package — which was written recently using modern Python technology — offers a WSGI wrapper around its JSON-RPC dispatcher, making it easy to incorporate into modern Python web application stacks, or to deploy stand-alone using the Standard Library's `wsgiref` package.

The server code is quite simple, as an RPC mechanism should be. As with XML-RPC, we merely need to name the functions that we want offered over the network, and they become available for queries. (You can also pass an object, and its methods will be registered with the server all at once.)

Listing 18–6. *JSON-RPC Client*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - jsonrpc_client.py
# JSON-RPC client

from lovely.jsonrpc import proxy
proxy = proxy.ServerProxy('http://localhost:7002')
print proxy.lengths((1,2,3), 27, {'Sirius': -1.46, 'Rigel': 0.12})
```

Writing client code is also quite simple. Sending several objects whose lengths we want measured — and having those data structures sent right back by the server — enables us to see several details about this particular protocol.

First, note that the protocol allowed us to send as many arguments as we wanted; it was not bothered by the fact that it could not introspect a static method signature from our function. This is similar to XML-RPC, but it is very different from XML-RPC mechanisms built for traditional, statically-typed languages.

Second, note that the `None` value in the server's reply passes back to us unhindered. This is because this value is supported natively by the protocol itself, without our having to activate any non-standard extensions:

```
$ python jsonrpc_server.py
Starting server
[In another command window:]
$ python jsonrpc_client.py
[[3, [1, 2, 3]], [None, 27], [2, {'Rigel': 0.12, 'Sirius': -1.46}]]
```

Third, note that there is only one kind of sequence supported by JSON-RPC, which means that the tuple sent by our client had to be coerced to a list to make it across.

Of course, the biggest difference between JSON-RPC and XML-RPC — that the data payload in this case is a small, sleek JSON message that knows natively how to represent each of our data types — is not even visible here. This is because both mechanisms do such a good job of hiding the network from our code. Running Wireshark on my localhost interface while running this example client and server, I can see that the actual messages being passed are as follows:

```
{"version": "1.1",
 "params": [[1, 2, 3], 27, {"Rigel": 0.12, "Sirius": -1.46}],
 "method": "lengths"}
{"result": [[3, [1, 2, 3]], [null, 27],
            [2, {"Rigel": 0.12, "Sirius": -1.46}]]}
```

Note that the popularity of JSON-RPC version 1 has led to several competing attempts to extend and supplement the protocol with additional features. You can do research online if you want to explore the current state of the standard and the conversation around it. For most basic tasks, you can simply use a good third-party Python implementation like `lovely.jsonrpc` and not worry about the debate over extensions to the standard.

It would be remiss of me to leave this topic without mentioning one important fact. Although the preceding example is synchronous — the client sends a request, then waits patiently to receive only a single response and does nothing useful in the meantime — the JSON-RPC protocol does support attaching `id` values to each request. This means you can have several requests underway before receiving any matching responses back with the same `id` attached. I will not explore the idea any further here because asynchrony, strictly speaking, goes beyond the traditional role of an RPC mechanism; (function calls in traditional procedural languages are, after all, strictly synchronous events. But if you find the idea interesting, you should read the standard and then explore which Python JSON-RPC libraries might support your need for supporting asynchrony.

Self-documenting Data

You have just seen that both XML-RPC and JSON-RPC appear to support a data structure very much like a Python dictionary, but with an annoying limitation. In XML-RPC, the data structure is called a *struct*, whereas JSON calls it an *object*. To the Python programmer, however, it looks like a dictionary, and your first reaction will probably be annoyance that its keys cannot be integers, floats, or tuples.

Let us look at a concrete example. Imagine that you have a dictionary of physical element symbols indexed by their atomic number:

```
{1: 'H', 2: 'He', 3: 'Li', 4: 'Be', 5: 'B', 6: 'C', 7: 'N', 8: 'O'}
```

If you need to transmit this dictionary over an RPC mechanism, your first instinct might be to change the numbers to strings, so that the dictionary can pass as a struct or object. It turns out that, in most cases, this instinct is wrong.

Simply put, the struct and object RPC data structures are not designed to pair keys with values in containers of an arbitrary size. Instead, they are designed to associate a small set of pre-defined attribute names with the attribute values that they happen to carry for some particular object. If you try to use a struct to pair random keys and values, you might inadvertently make it very difficult to use for people unfortunate enough to be using statically-typed programming languages.

Instead, you should think of dictionaries being sent across RPCs as being like the `__dict__` attributes of your Python objects, which — if you are an experienced Python programmer — you should generally *not* find yourself using to associate an arbitrary set of keys with values! Just as your Python objects tend to have a small collection of attribute names that are well-known to your code, the dictionaries you send across RPC should associate a small number of pre-defined keys with their related values.

All of this means that the dictionary that I showed a few moments ago should actually be serialized as a list of explicitly labelled values if it is going to be used by a general-purpose RPC mechanism:

```
{{'number': 1, 'symbol': 'H'},
 {'number': 2, 'symbol': 'He'},
 {'number': 3, 'symbol': 'Li'},
 {'number': 4, 'symbol': 'Be'},
 {'number': 5, 'symbol': 'B'},
 {'number': 6, 'symbol': 'C'},
 {'number': 7, 'symbol': 'N'},
 {'number': 8, 'symbol': 'O'}}
```

Note that the preceding examples show the Python dictionary as you will pass it into your RPC call, not the way it would be represented on the wire.

The key difference in this approach (besides the fact that this dictionary is appallingly longer) is that the earlier data structure was meaningless unless you knew ahead of time what the keys and values meant; it relied on convention to give the data meaning. But here we are including names with the data and that makes this example self-descriptive to some extent; that is, someone looking at this data on the wire or in his program has a higher chance of guessing what it represents.

I will not argue that this is always a good idea. I am merely pointing out that this is how both XML-RPC and JSON-RPC expect you to use their key-value types, and that this is where the names *struct* and *object* came from. They are, respectively, the C language and JavaScript terms for an entity that holds named attributes. Again, this makes them much closer to being like Python objects than Python dictionaries.

If you have a Python dictionary like the one we are discussing here, you can turn it into an RPC-appropriate data structure, and then change it back with code like this:

```
>>> elements = {1: 'H', 2: 'He'}
>>> t = [ {'number': key, 'symbol': elements[key]} for key in elements ]
>>> t
[{'symbol': 'H', 'number': 1}, {'symbol': 'He', 'number': 2}]
>>> dict( (obj['number'], obj['symbol']) for obj in t )
{1: 'H', 2: 'He'}
```

Using named tuples (as they exist in the most recent versions of Python) might be an even better way to marshal such values before sending them if you find yourself creating and destroying too many dictionaries to make this transformation appealing.

Talking About Objects: Pyro and RPyC

If the idea of RPC was to make remote function calls look like local ones, then the two basic RPC mechanisms we have looked at actually fail pretty spectacularly. If the functions we were calling happened to only use basic data types in their arguments and return values, then XML-RPC and JSON-RPC would work fine. But think of all of the occasions when you use more complex parameters and return values instead! What happens when you need to pass live objects?

This is generally a very hard problem to solve for two reasons.

First, objects have different behaviors and semantics in different programming languages. Thus mechanisms that support objects tend to either be restricted to one particular language, or they tend to offer an anemic description of how an "object" can behave that is culled from the lowest common denominator of the languages it wants to support.

Second, it is often not clear how much state needs to travel with an object to make it useful on another computer. True, an RPC mechanism can just start recursively descending into an object's attributes and getting those values ready for transmission across the network. However, on systems of even moderate complexity, you can wind up walking most of the objects in memory by doing simple-minded recursion into attribute values. And having gathered up what might be megabytes of data for transmission, what are the chances that the remote end actually needs all of that data?

The alternative to sending the entire contents of every object passed as a parameter, or returned as a value, is to send only an object name that the remote end can use to ask questions about the object's attributes if it needs to. This means that just one item out of a highly connected object graph can be quickly transmitted, and only those parts of the graph that the remote site actually needs wind up getting transmitted.

However, both schemes often result in expensive and slow services, and they can make it very difficult to keep track of how one object is allowed to affect the answers provided by another service on the other end of the network.

In fact, the task that XML-RPC and JSON-RPC forces upon you — the task of breaking down the question you want to ask a remote service so simple data types can be easily transmitted — often winds

up being, simply, the task of software architecture. The restriction placed on parameter and return value data types makes you think through your service to the point where you see exactly what the remote service needs and why. Therefore, I recommend against jumping to a more object-based RPC service simply to avoid having to design your remote services and figure out exactly what data they need to do their job.

There are several big-name RPC mechanisms like SOAP and CORBA that, to varying degrees, try to address the big questions of how to support objects that might live on one server while being passed to another server on behalf of a client program sending an RPC message from yet a third server. In general, Python programmers seem to avoid these RPC mechanisms like the plague, unless a contract or assignment specifically requires them to speak these protocols to another existing system. They are beyond the scope of this book; and, if you need to use them, you should be ready to buy at least an entire book on each such technology — they can be that complex!

But when all you have are Python programs that need to talk to each other, there is at least one excellent reason to look for an RPC service that knows about Python objects and their ways: Python has a number of very powerful data types, so it can simply be unreasonable to try "talking down" to the dialect of limited data formats like XML-RPC and JSON-RPC. This is especially true when Python dictionaries, sets, and `datetime` objects would express exactly what you want to say.

There are two Python-native RPC systems that we should mention: *Pyro* and *RPyC*.

The Pyro project lives here: `http://ww.xs4all.nl/~irmen/pyro3/`

This well-established RPC library is built on top of the Python `pickle` module, and it can send any kind of argument and response value that is inherently *pickle-able*. Basically, this means that, if an object) and its attributes) can be reduced to its basic types, then it can be transmitted. However, if the values you want to send or receive are ones that the `pickle` module chokes on, then Pyro will not work for your situation.

You should also check out the `pickle` documentation in the Standard Library. This library includes instructions on making classes pickle-able if Python cannot figure out how to pickle them itself.

An RPyC Example

The RPyC project lives here: `http://rpyc.wikidot.com/`

This project takes a much more sophisticated approach toward objects. Indeed, it is more like the approach available in CORBA, where what actually gets passed across the network is a reference to an object that can be used to call back and invoke more of its methods later if the receiver needs to. The most recent version also seems to have put more thought into security, which is important if you are letting other organizations use your RPC mechanism. After all, if you let someone give you some data to un-pickle, you are essentially letting them run arbitrary code on your computer!

You can see an example client and server in Listings 18–7 and 18–8. If you want an example of the incredible kinds of things that a system like RPyC makes possible, you should study these listings closely.

Listing 18–7. *An RPyC Client*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - rpyc_client.py
# RPyC client

import rpyc

def noisy(string):
»   print 'Noisy:', repr(string)

proxy = rpyc.connect('localhost', 18861, config={'allow_public_attrs': True})
fileobj = open('testfile.txt')
```

```
linecount = proxy.root.line_counter(fileobj, noisy)
print 'The number of lines in the file was', linecount
```

At first the client might look like a rather standard program using an RPC service. After all, it calls a generically-named `connect()` function with a network address, and then accesses methods of the returned proxy object as though the calls were being performed locally. However, if you look closer, you will see some startling differences! The first argument to the RPC function is actually a live file object that does not necessarily exist on the server. And the other argument is a function, another live object instead of the kind of inert data structure that RPC mechanisms usually support.

The server exposes a single method that takes the proffered file object and callable function. It uses these exactly as you would in a normal Python program that was happening inside a single process. It calls the file object's `readlines()` and expects the return value to be an iterator over which a `for` loop can repeat. Finally, the server calls the function object that has been passed in without any regard for where the function actually lives (namely, in the client). Note that RPyC's new security model dictates that, absent any special permission, it will only allow clients to call methods that start with the special prefix, `exposed_`.

Listing 18–8. *An RPyC Server*

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 18 - rpyc_server.py
# RPyC server

import rpyc

class MyService(rpyc.Service):
    def exposed_line_counter(self, fileobj, function):
        for linenum, line in enumerate(fileobj.readlines()):
            function(line)
        return linenum + 1

from rpyc.utils.server import ThreadedServer
t = ThreadedServer(MyService, port = 18861)
t.start()
```

It is especially instructive to look at the output generated by running the client, assuming that a small `testfile.txt` indeed exists in the current directory and that it has a few words of wisdom inside:

```
$ python rpyc_client.py
Noisy: 'Simple\n'
Noisy: 'is\n'
Noisy: 'better\n'
Noisy: 'than\n'
Noisy: 'complex.\n'
The number of lines in the file was 5
```

Equally startling here are two facts. First, the server was able to iterate over multiple results from `readlines()`, even though this required the repeated invocation of file-object logic that lived on the client. Second, the server didn't somehow copy the `noisy()` function's code object so it could run the function directly; instead, it repeatedly invoked the function, with the correct argument each time, on the client side of the connection!

How is this happening? Quite simply, RPyC takes exactly the opposite approach from the other RPC mechanisms we have looked at. Whereas all of the other techniques try to serialize and send as much information across the network as possible, and then leave the remote code to either succeed or fail with no further information from the client, the RPyC scheme only serializes completely immutable items

such as Python integers, floats, strings, and tuples. For everything else, it passes across an object name that lets the remote side reach back into the client to access attributes and invoke methods on those live objects.

This approach results in quite a bit of network traffic. It can also result in a significant delay if lots of object operations have to pass back and forth between the client and server before an operation is complete. Tweaking security properly is also an issue. To give the server permission to call things like `readlines()` on the client's own objects, I chose to make the client connection with a blanket assertion of `allow_public_attrs`. But if you are not comfortable giving your server code such complete control, then you might have to spend a bit of time getting the permissions exactly right for your operations to work without exposing too much potentially dangerous functionality.

So the technique can be expensive, and security can be tricky if the client and server do not trust each other. But when you need it, there is really nothing like RPyC for letting Python objects on opposite sides of a network boundary cooperate with each other. You can even let more than two processes play the game; check out the RPyC documentation for more details!

The fact that RPyC works successfully like this against vanilla Python functions and objects, without any requirement that they inherit from or mix in any special network capabilities, is an incredible testimonial to the power that Python gives us to intercept operations performed on an object and handle those events in our own way — even by asking a question across the network!

RPC, Web Frameworks, Message Queues

Be willing to explore alternative transmission mechanisms for your work with RPC services. The classes provided in the Python Standard Library for XML-RPC, for example, are not even used by many Python programmers who need to speak that protocol. After all, one often deploys an RPC service as part of a larger web site, and having to run a separate server on a separate port for this on particular kind of web request can be quite annoying.

There are three useful ways that you can look into moving beyond overly simple example code that makes it look as though you have to bring up a new web server for every RPC service you want to make available from a particular site.

First, look into whether you can use the pluggability of WSGI to let you install an RPC service that you have incorporated into a larger web project that you are deploying. Implementing both your normal web application and your RPC service as WSGI servers beneath a filter that checks the incoming URL enables you to allow both services to live at the same hostname and port number. It also lets you take advantage of the fact that your WSGI web server might already provide threading and scalability at a level that the RPC service itself does not provide natively.

Putting your RPC service at the bottom of a larger WSGI stack can also give you a way to add authentication if the RPC service itself lacks such a facility. See Chapter 12 for more information about WSGI.

Second, instead of using a dedicated RPC library, you may find that your web framework of choice already knows how to host an XML-RPC, JSON-RPC, or some other flavor of RPC call. This means that you can declare RPC endpoints with the same ease that your web framework lets you define views or RESTful resources. Consult your web framework documentation and do a web search for RPC-friendly third-party plug-ins to see whether this is possible in your case.

Third, you might want to try sending RPC messages over an alternate transport that does a better job than the protocol's native transport of routing the calls to servers that are ready to handle them. Message queues, which are discussed in Chapter 8, are often an excellent vehicle for RPC calls when you want a whole rack of servers to stay busy sharing the load of incoming requests. If you explore a well-designed RPC library like `lovely.jsonrpc`, you will find that you can define your own transports that send and receive information through your own mechanism of choice, rather than insisting that HTTP to a fixed IP address be used.

Recovering From Network Errors

Of course, there is one reality of life on the network that RPC services cannot easily hide: the network can be down or even go down in the middle of a particular RPC call.

You will find that most RPC mechanisms simply raise an exception if a call is interrupted and does not complete. Note that an error, unfortunately, is no guarantee that the remote end did not process the request — maybe it actually did finish processing it, but then the network went down right as the last packet of the reply was being sent. In this case, your call would have technically happened and the data would have been successfully added to the database or written to a file or whatever the RPC call does. However, you will think the call failed and want to try it again — possibly storing the same data twice.

Fortunately, there are a few tricks you can use when writing code that delegates some function calls across the network.

First, be careful to distinguish exceptions in the remote code from problems with the protocol and network itself. The former often have to be fatal errors, whereas the latter can sometimes be recovered from by re-trying automatically; good RPC libraries will use different Python exceptions for these two cases, so that you can easily distinguish between them.

Second, take this advice offered in Chapter 5: instead of littering your code with a `try…except` everywhere that an RPC call is made, try wrapping larger pieces of code that have a solid semantic meaning and can more cleanly be re-attempted or recovered from. If you guard each and every call with an exception handler, after all, you will have lost most of the benefit of RPC: that your code is supposed to be convenient to write, and not make you constantly attend to the fact that function calls are actually being forwarded over the network! In cases where you decide your program should re-try a failed call, you might want to try using something like the exponential back-off algorithm you saw for UDP in Chapter 3. This approach lets you avoid hammering an overloaded service and making the situation worse.

Finally, be careful about working around the loss of exception detail across the network. Unless you are using a Python-aware RPC mechanism, you will probably find that what would normally be a familiar and friendly `KeyError` or `ValueError` on the remote side becomes some sort of RPC-specific error whose text or numeric error code you have to inspect in order to have any chance of telling what happened.

Binary Options: Thrift and Protocol Buffers

The first two RPC mechanisms we looked at were textual: XML-RPC and JSON-RPC require raw data to be turned into strings for transmission, and then parsed and decoded again on the remote end. The Python-specific systems that we then discussed supported less verbose forms of data interchange, but without any ability to operate between different programming languages.

It is possible you will want both features: a compact and efficient binary format *and* support across several different languages. Here are a few options:

- Some JSON-RPC libraries support the BSON protocol, which provides a tight binary transport format and also an expanded range of data types beyond those supported by JSON.
- The Apache Foundation is now incubating Thrift, an RPC system developed several years ago at Facebook and released as open source. The parameters and data types supported by each service and method are pre-defined in files that are then shared by the developers programming the clients and servers.

- Google Protocol Buffers are popular with many programmers, but strictly speaking they are not a full RPC system; instead, they are a binary data serialization protocol. At the time of writing, Google has not released the additional pieces that they have written on top of Protocol Buffers to support a full-fledged RPC round-trip. To perform an actual remote procedure call, you might have to roll your own convention.

And, of course, systems that I have never even heard of — and perhaps some that have not yet been invented — will come in to vogue over the years that this book is in print. But whatever RPC system you deploy, the basic principles discussed here should help you use it effectively.

Summary

Remote procedure calls let you write what look like normal Python function calls that actually reach across the network and call a function on another server. They do this by serializing the parameters so that they can be transmitted; they then do the same with the return value that is sent back.

All RPC mechanisms work pretty much the same way: you set up a network connection, and then make calls on the proxy object you are given in order to invoke code on the remote end. The old XML-RPC protocol is natively supported in the Python Standard Library, while good third-party libraries exist for the sleeker and more modern JSON-RPC.

Both of these mechanisms allow only a small handful of data types to pass between the client and server. If you want a much more complete array of the Python data types available, then you should look at the Pyro system, which can link Python programs together across the network with extensive support for native Python types. The RPyC system is even more extensive, and it allows actual objects to be passed between systems in such a way that method calls on those objects are forwarded back to the system on which the object actually lives.

Index

B

C

E

I

J

K

Q

R

T

V

W

X, Y

Z